THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 62, No. 3, March 1983
Printed in U.S.A.

Traffic Service Position System No. 1B:

Real-Time Architecture Utilizing the DMERT Operating System

By R. J. GILL, G. J. KUJAWINSKI, and E. H. STREDDE

(Manuscript received June 30, 1982)

The Traffic Service Position System No. 1B (TSPS No. 1B) architecture was conceived to increase performance significantly for future features and traffic growth. The design preserves the TSPS No. 1 software with minimal changes. At the same time, the 3B20 Duplex Processor (3B20D) used in TSPS No. 1B provides additional processor peripherals and a modern programming environment with a real-time operating system. This paper describes how the TSPS No. 1 software, initially designed to run on the Stored Program Control No. 1A (SPC 1A), executes on the SPC 1B of the TSPS No. 1B. The SPC 1B is a 3B20D tailored for the TSPS application and provides an SPC 1A environment by directly emulating its instruction set. The paper also presents major TSPS application processes and their interaction with the emulated TSPS process and the Duplex Multi-Environment Real-Time (DMERT) operating system of the SPC 1B. In addition, the paper describes the integration of TSPS maintenance software into the DMERT maintenance structure.

I. INTRODUCTION

The Traffic Service Position System No. 1B (TSPS No. 1B) real-time architecture was designed to meet the project goals discussed in Ref. 1. The implementation of this architecture entailed four major developments:

(*i*) Replacement of the Stored Program Control No. 1A (SPC 1A) of TSPS No. 1 with the 3B20D Processor, the TSPS Peripheral System Interface (PSI), and microcode to execute the SPC 1A instruction set,

which together comprise the Stored Program Control No. 1B (SPC 1B)

(*ii*) Emulation of most existing TSPS No. 1 software structured as a process under the Duplex Multi-Environment Real-Time (DMERT) operating system

(*iii*) Development of additional processes to support the emulation

(*iv*) Integration of the PSI and TSPS peripheral maintenance into the overall DMERT maintenance structure.

Before discussing the TSPS No. 1B real-time architecture, this paper presents two sections of background information. Section II presents an overview of how TSPS operates using the SPC 1A. Section III reviews the fundamentals of DMERT, and Section IV describes the SPC 1B and the TSPS No. 1B software architectures. These sections enable the reader to understand the real-time architecture of the TSPS No. 1B. More detailed information can be obtained by reading the references.

II. TSPS NO. 1 REAL-TIME ARCHITECTURE

This section presents a brief overview of the TSPS No. 1 operation on the SPC 1A and provides a base for understanding how the TSPS No. 1 was emulated on the TSPS No. 1B. A complete description of TSPS No. 1 operation on the SPC 1A can be found in Refs. 2 and 3.

2.1 SPC 1A programming environment

The SPC 1A uses 20-bit addresses to reference approximately one million 20-bit words of main memory. The address spectrum consists of up to 30 store name codes with each name code containing 32K 20-bit words. Store name code 0 is not used since low memory addresses are mapped into the SPC 1A's buffer bus (see Section 2.2). Store name code 31 (037) is not implemented. Hence, the maximum physical memory size for the SPC 1A is 960K 20-bit words. The SPC 1A does not support memory management. Hence, there is no virtual addressing; all addresses are physical addresses. All of memory is equally accessible (shared) by all programs.

Write protection can be set on a 2K word boundary within a name code. Protected areas within the name code, however, must be contiguous, and there can be only one protection change boundary within a name code. Memory is unprotected from the high end of the address spectrum within each name code. For example, if one fourth of a name code is to be unprotected (read/write), then the first three quarters would be read only, and the last quarter would be unprotected. Protected areas can be unlocked to change programs or fixed data by having the processor execute a special unlocking sequence. Typically, protected areas are used for office data, read-only tables, and program

logic. Unprotected areas are used for volatile data (e.g., call information).

SPC 1A instructions are 40 bits wide (two 20-bit words). The odd-addressed word contains the operation code, while addresses or data, when present, are placed in the even-addressed word. There are relatively few instructions, but a single instruction can have many options and perform several operations. Each instruction takes from one to three 6.3-microsecond machine cycles to execute. The execution time of each instruction is fixed regardless of the number of options specified. Hence, the time a segment of code will execute can be precisely determined.

2.2 Buffer bus

The SPC 1A buffer bus is an internal collection of processor and peripheral status and control registers. The buffer bus registers are used for such things as peripheral and processor configuration control and status indications, interrupt sources, and interrupt masks. The buffer bus consists of 24 registers with 20 to 24 bits each. Each register has a low-memory address (e.g., 600 or 2200) associated with it. All buffer bus registers can be read with Memory to Register (MR) instructions. Some registers can be written, set, or reset using Register(s) to Memory (RM or RRM) instructions or Constant to Buffer Bus (CBB) instructions. Other registers are read-only.

2.3 Interrupt structure

2.3.1 Interrupt levels

There are nine interrupt levels in the SPC 1A. These interrupt levels are A (highest priority), B, C, E, F, G, H, J, and K. A tenth level, L (commonly referred to as base level), runs continuously in the absence of any interrupt. Normally, base level is only interrupted every 5 ms by J-level. Although, base level is the lowest-priority processing level, the bulk of TSPS software (e.g., call processing, diagnostics, audits) executes in base level.

Each interrupt level can only interrupt lower-priority levels with the sole exception being that A- and B-level can interrupt each other. An A-level interrupt is caused either by a manual action at the Maintenance Control Center (MCC) Control and Display (CD) frame or by the execution of an ANOP instruction (used to fill unused instruction space). B-level is entered as a result of a processor switch or for emergency actions required as a result of system-sanity-check failures. C- and K-level interrupts generally occur as a result of processor errors. C-level handles SPC 1A Processor fault recovery. K-level interrupts are only enabled during processor diagnostics for error-data recording. SPC 1A store errors will generate E- and G-level interrupts.

E-levels also result from software errors such as invalid addresses or protection violations. E-levels provide data for immediate problem analysis and necessary store reconfiguration. G-level is used for collecting store-error data for intermittent failures. Peripheral-unit errors will generate an F-level interrupt. F-level software is dedicated to peripheral fault recovery. J- and H-levels perform the system I/O.

2.3.2 J- and H-level interactions

The J-level interrupt is generated every 5 milliseconds by a clock pulse. Because of its periodic nature, J-level provides the main time reference for all application processing. Within J-level there are two classes of jobs: high priority and low priority. The high-priority jobs are executed first. While they are running, H-level is inhibited. In making the transition to low-priority jobs, H-level is enabled. H-level programs consist of the high-priority, J-level jobs. An H-level is caused whenever J-level executes longer than 5 ms and low-priority jobs are being run (see Fig. 1*). This ensures that the high-priority jobs are executed every 5 ms, unless H-level runs longer than 5 ms. In this case the high-priority jobs would miss an execution, as H-level cannot interrupt itself.

J- and H-levels on the SPC 1A have two separate interrupt sources that are driven by the same clock pulse. The H-level interrupt source is only enabled when low-priority J-level is executing. When a return from interrupt is performed from low-priority J-level, the H-level inhibit bit is set.

2.3.3 Interrupt handling

When an interrupt occurs under normal conditions on the SPC 1A, the processor saves the address of the interrupted program and passes control to the first instruction of the interrupt program for that level. The address of the interrupt program and the save area (bin) for the interrupted program address are known (i.e., hard-wired) by the processor. The interrupt-handling program for each interrupt, except J-level, first saves the contents of the seven general-purpose registers in memory and then determines what actions to take. Because of the frequency of the J-level interrupt (every 5 ms), general-register contents are copied to an auxiliary set of registers by the hardware. The overhead of saving and restoring the registers every 5 ms is greatly reduced by performing this function in hardware.

The return from interrupt is normally performed by the Execute Go Back To Normal (EGBN) instruction. The one exception is J-level. J-

* Acronyms and abbreviations used in the figures and text are defined in the Glossary.

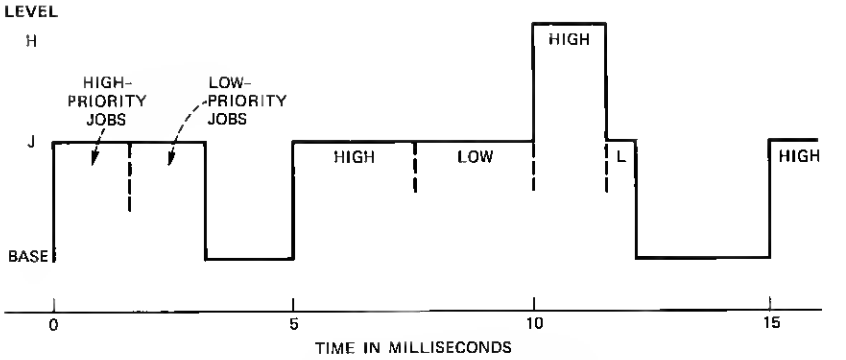

Fig. 1—5-ms input/output processing.

level uses the Go Back To Normal H- or J-level (GBNHJ) instruction. In returning, the interrupt handler (except for J-level) first restores the general-purpose register contents from memory (if it were going to return to the point of interrupt) and then executes the EGBN. The EGBN instruction determines what interrupt level is being returned from by looking at the interrupt-level activity flags in the buffer bus. It restores the address of the interrupted program from the save area for that interrupt and clears the highest interrupt bit in the buffer bus interrupt-level activity word, thus dropping to the next highest, previously active level. If an intermediate-level interrupt is pending, it would be serviced at this time. The GBNHJ is slightly different in that it causes the general-purpose registers to be restored by hardware from the auxiliary set of registers. It also disables the H-level interrupt in addition to performing the functions of the EGBN.

Any interrupt servicing routine can change its point of return by overwriting the saved address in the appropriate interrupt bin in memory. Also, an interrupt is effectively returned from by merely clearing its activity bit in the buffer bus, which essentially erases the history of the interrupt. This immediately puts the program into the next lower, previously active level. This re-enables that interrupt level as well as any intervening levels.

2.3.4 Interrupt inhibiting

Interrupts are normally only masked when the program is handling a higher-priority interrupt. The interrupt activity bits in the buffer bus inhibit lower-priority interrupts from occurring. E-, F-, H-, and J-level interrupts can also be individually inhibited by setting the specific inhibit bits in the buffer bus. The H-level inhibit should always be set except when in low-priority J-level. The J-level inhibit can be set for brief periods when executing "critical region" code in base level, which

should not be interrupted by J-level activity because of potential interference. Similarly, the H-level inhibit could be set in low-priority J-level to prevent H-level interference. Some instructions have an "inhibit I/O interrupt" option that inhibits H- and J-level interrupts until the completion of the following instruction. For some instructions this option is implicit (not an option, but always on). The E- and F-level inhibits can be set manually as well as by software. C-level interrupts can be inhibited by software, and A- and B-level interrupts can be manually inhibited from the MCC.

2.4 Program structure

J-level (and H-level) programs execute under control of the Executive Control for I/O (ECIO) program. For the most part ECIO passes control to a predetermined and fixed set of programs every 5 ms. The list of jobs varies from execution to execution, but is cyclical over a 300-ms interval for high-priority work and over a 200-ms interval for low-priority work. Thus, J-level jobs execute with frequencies that are multiples of 5 ms, ranging up to 200 or 300 ms. Some program executions are permanent (always active) while others are run only on demand.

Similar to J-level, base-level programs are run under control of the Executive Control for the Main Program (ECMP) routines. Base-level programs execute in one of six priority classes. These priority classes are interject (highest priority), A, B, C, D, and E. ECMP passes control to jobs in priority classes A through E with a relative frequency of 15:8:4:2:1, respectively (see Fig. 2). Jobs within each priority class can run every execution of the priority class or only as needed on demand. Each priority class has a set of task dispensers that pass control to individual programs (tasks) when work is to be performed.

Interject work is requested when certain immediate actions are required (e.g., to immediately unload a full input hopper) or at a fixed frequency on demand (i.e., J-level-initiated sanity check of interject operation). A check for interject requested work is performed at least once in every base-level loop (one E-priority class to E-priority class cycle) and after every task. Interject should be serviced quickly. Hence, all base-level tasks are designed to run in short segments. Figure 3 illustrates the base-level priority-class execution with the fixed-interject check and another representative interject job executing in the middle of a priority class.

2.5 System integrity

The objective of system integrity is to provide an uninterrupted call-processing environment. The reliability goal for TSPS No. 1 is to achieve less than three minutes per year of total system outage

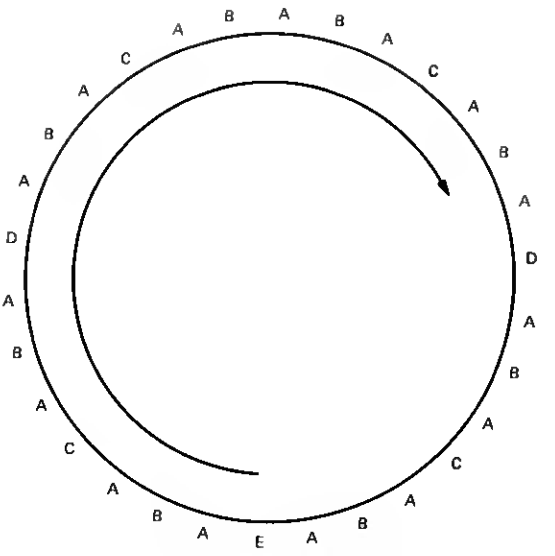

Fig. 2—TSPS No. 1 base-level loop.

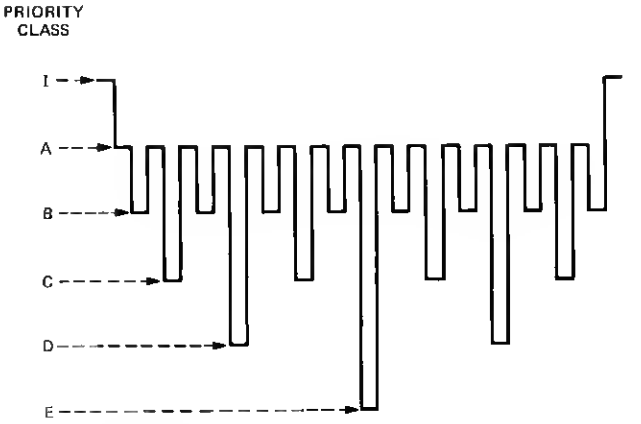

Fig. 3—TSPS No. 1 base-level, priority-class frequency.

averaged over all systems. This goal covers outages attributable to any possible cause. Hence, the term "system integrity" is used in this and other sections of this paper to encompass software stability as well as the traditional hardware reliability.

2.5.1 Hardware integrity

The maintenance strategy for TSPS hardware is based on the duplication of all critical units. This hardware redundancy allows

faulty units to be switched out of service with the load being carried by the remaining good unit. Maintenance programs are organized on priority levels such that the faulty unit can be removed from service as soon as possible, and then later, while the system is processing calls, the faulty unit is diagnosed in order to isolate the failing circuit pack. There are three main types of maintenance programs for the TSPS peripherals. These are, in order of decreasing priority, fault-recognition programs, diagnostic programs, and exercise programs.

Fault-recognition programs run when the presence of a fault is detected. They determine which of the duplicated units is in error, and reconfigure the system around the problem. Before returning to call processing, the fault-recognition program initiates a diagnostic request on the unit suspected of malfunctioning. The purpose of the diagnostic program is to provide resolution of the fault by indicating to the craft personnel the smallest replaceable unit (e.g., a circuit pack). This is accomplished by running a series of tests on the suspected hardware unit and then comparing the actual test results with a set of expected values. Another method of detecting faults employs the use of exercises. These programs are similar to diagnostics in that they run selected tests on the hardware. They differ in that they are intended to find faults in circuits not exercised by normal system operation (e.g., by call processing). Unlike diagnostics that are initiated by fault-recognition programs upon detection of a fault, the exercise programs are scheduled periodically.

2.5.2 Software integrity

2.5.2.1 Initialization and recovery. Whenever the state of the software is such that normal processing cannot continue, call-processing recovery actions are taken in an attempt to restore the system's sanity. The least severe actions are taken first. If these fail, the recovery attempt is escalated to the next highest level. The five TSPS recovery phases are Minor Audits, Major Audits, Selected Audits (miniphases), System Initialization A (SIA) and System Initialization B (SIB). Except for Selected Audit phases, these recovery phases can be manually requested or automatically generated by the software. Selected Audits phases can only be initiated by software, but they can be manually inhibited.

All audits that run during a recovery phase are "stitched" together. This means that they are run consecutively. Meanwhile, the normal base-level priority-class execution and all call processing is temporarily suspended. Minor Audit phases are short and, as a result, they have the least effect on call-processing activity. Major Audit phases are more extensive, and, hence, have a more disturbing effect. Selected Audit phases are run as the result of problems detected by software

sanity checks. A specific set of audits are stitched together, depending on the error found. SIAs and SIBs run a complete set of audits. An SIA also zeroes most unprotected memory and initializes the hardware. An SIB is more extensive in that it performs a more thorough hardware initialization. If an SIB fails to restore system sanity, another SIB will automatically be taken with a different hardware configuration. Looping SIBs with hardware reconfigurations will continue indefinitely until the system is recovered or manual intervention takes place. For the Minor, Major, and SIA phases, recovery actions are identical whether the phase is software generated or manually requested. On the other hand, a manually requested SIB will zero all unprotected memory, but it will not cause a hardware reconfiguration.

2.5.2.2 Reference returns. Under certain conditions, maintenance-interrupt routines (levels A through K) must return to H-level, J-level, or base-level at a reference point in the job administration stream when a return to the interrupted point is not warranted. This safe transfer of control is called a reference return. The base-level reference return will result in the base-level cycle restarting at the end of E priority. The H- and J-level reference returns result in the cancellation of the job being run at the time of the interrupt, but the remaining scheduled jobs are executed.

2.5.2.3 Sanity. Sane program execution is monitored via a hierarchical scheme. Base level checks itself every E-priority class by determining that the various priority classes have been run the proper relative number of times (15:8:4:2:1). Base-level sanity is checked in J-level by requesting an interject job every 500 ms and monitoring its execution. Failure to execute interject suggests a base-level loop. High-priority J-level (and, hence, H-level) monitors low-priority J-level by monitoring the execution of a fixed 100-ms job. High-priority J-level (and H-level) has the responsibility to reset a hardware sanity timer every 500 ms. Failure to reset the timer within 640 ms or resetting it too soon (less than 320 ms) will cause the timer to generate a B-level interrupt. Low-priority J-level insanity will result in a minor audit call-processing phase. Continued interject response failures or base-level priority class execution insanity will also trigger a minor phase. High-priority J-level also monitors system sanity by checking the average time of the last three E-E cycles. If this time exceeds a threshold a minor phase is taken. Lower threshold crossings will trigger overload recovery actions.

2.5.2.4 Overload strategy. If at any point the elapsed time to run the last three E-E cycles exceeds a minimum threshold, phase 1 overload actions are taken. These actions consist of gradually busying trunks back to the local offices and reducing the rate at which processing of new calls is allowed to begin. If during phase 1 overload the elapsed E-E times exceed another, higher threshold, phase 2 actions are taken.

These actions busy all trunks to the local offices except for a minimum number and inhibit the processing of any new calls in the system. As E-E times return to acceptable levels, the system returns to phase 1 actions and eventually to normal. The return to normal operation (unbusying trunks and increasing the new call-processing rate) is done gradually as the overload subsides.

III. DMERT OPERATING SYSTEM

The DMERT operating system[4] evolved from the MERT[5] operating system. While MERT was designed to operate on a simplex minicomputer, DMERT has incorporated maintenance software to control the duplex hardware in order to provide Electronic Switching System (ESS) reliability on the 3B20D Processor.

3.1 Processes

An executable entity under DMERT is called a process. A process is a collection of programs and data with a distinct virtual address space (see Section 3.2) that is executed as a unit to perform a single (or set of related) function(s). Once in execution, a process controls the scheduling of its internal routines, barring any external stimulus such as an interrupt or fault. While the process is executing it appears to have an entire (virtual) machine to itself, although it may he interrupted by higher-priority processes or even swapped out to disk while waiting for some event to occur (e.g., the completion of an I/O request). The process address space may be completely protected from access by other processes or it may be shared at will. One process can communicate with another via several mechanisms supported by DMERT, as described in Section 3.6.

3.2 Memory management

The basic unit of memory handled by tbe 3B20D's memory management system is a page: 2K 8-bit bytes (five bundred twelve 32-bit words). A segment consists of 1 to 64 pages that need not be contiguous in physical memory. A process in DMERT consists of at least three segments, where one segment contains the process stack, another contains the process control block (PCB), and at least one segment contains executable code. Each process executes in its own logical (or virtual) address space, which may be as large as 16M bytes (4M words). Memory management swaps processes between memory and disk, enabling many processes to coexist even though the sum of their memory requirements exceeds the physical memory of the processor. It also provides protection from misuse (i.e., writing into read-only memory) and unauthorized access by other processes.

The information required to perform virtual address to physical

address translation is maintained by DMERT in segment and page tables in memory. Accessing these tables for every address translation would take a considerable amount of time (6.8 microseconds each). To speed up this process a high-speed associative memory called an address translation buffer (ATB) is used to keep the most recently used translation data. There are eight 64- × 2-word-set associative memories. Four of the ATBs can be dedicated to individual processes. One other is dedicated to the kernel. The other three are shared dynamically by all other processes. There is one each for kernel, supervisor, and user processes. These processes are described in Section 3.3. Address translation via the ATB is performed in 150 nanoseconds.

3.3 Abstract machines

DMERT supports four levels of software. Each level provides a different abstract view of the machine to the software. These abstract machine levels are:

(*i*) The kernel
(*ii*) Kernel processes
(*iii*) Supervisor processes
(*iv*) User processes.

The kernel is the lowest abstract software level under DMERT and the core of the operating system. It provides the basic services of the operating system, such as interrupt control, process dispatching, scheduling, and timing. It essentially extends the set of operations for kernel and supervisor processes.

The kernel-process level is used for those processes that have stringent timing constraints and must respond rapidly to real-time stimuli such as interrupts. Also, processes that must directly interact with hardware devices (such as a peripheral-unit driver) are coded as kernel processes. Similar to the kernel, kernel processes can have direct hardware access. Kernel processes also share some system data (e.g., the kernel stack and message buffers) with the kernel. Other system data are accessed via kernel services. Because of their performance requirements, kernel processes are totally memory resident and cannot be swapped out to disk. Some DMERT kernel processes (e.g., the process manager and memory manager) are referred to as special processes and share the kernel address space.

The next highest abstract machine level is for supervisor processes. This level is for those processes that neither have stringent timing constraints nor require direct access to the hardware or system data. Access to hardware (i.e., for I/O) and to system data is provided to supervisor processes by the kernel and kernel processes. The hardware and system data are completely shielded from supervisor processes.

Because of this layered software structure, errors in supervisor processes are much less likely to have catastrophic system effects. The price for this protection, however, is slower response time and performance. To improve response time supervisor processes have the option of being memory resident and not being swapped out to disk.

Both kernel and supervisor processes have the option of being nonkillable. A nonkillable process must perform its own internal fault recovery. A killable process can be terminated and recreated under certain error conditions.

The highest and most abstract software level under DMERT is the user-process level. User processes exist only in conjunction with a supervisor process, and in effect are just a unique state of that supervisor. The user portion of the process, however, does have a separate virtual address space distinct from the supervisor portion. A supervisor process can gain access to its user address space, but the reverse is not true. As a result, user processes are totally removed from the details of the actual machine and operating system under which they execute. Hence, the user level is the easiest programming level. However, user processes have poorer performance than supervisor processes.

3.4 Interrupt structure

Interrupts are detected between the execution of two instructions and change the sequence of execution. More specifically, an interrupt results in the interruption of the current executing process and a transfer of control to a specific interrupt-handling process. The state of the interrupted process is saved on the interrupt stack so that it can be restored at the completion of the interrupt processing and the interrupted process can resume execution. There are 32 maskable, hardware interrupt sources contained in the interrupt source (IS) register. The 3B20D also has four unmaskable interrupt sources. Interrupts can be generated by hardware (i.e., clocks and peripheral devices), microcode, and software. Corresponding to the IS is an interrupt mask (IM) register. The IM and IS registers are "anded" together to determine which interrupts are allowed to occur between any two instructions.

Table I shows the layout of the IS as used in TSPS No. 1B. Bit 0 is the highest-priority interrupt source. That is, if more than one interrupt source is set and unmasked, the lowest-order bit position will be serviced first. Of particular interest are the Program Interrupt Request (PIR) sources (bits 17 through 31). There is one PIR per DMERT execution level 1 through 15 (see below). A PIR is set in response to a process request to send an event or fault (discussed later). The PIR corresponding to the execution level of the receiving process is set.

Table I—DMERT interrupt source register

Bit	Use
0–1	Hardware errors
2	Software errors
3	Unused
4	PSI errors
5	Timer (10 ms)
6	TSPS (5 ms)
7–9	Unused
10–13	Direct Memory Access Input/Output (DMA I/O)
14–16	Non-DMA I/O
17–31	Program Interrupt Requests (PIRs)

Since supervisor and user processes always receive their events at level 1, PIR 0 is not required. For user processes, the associated supervisor process receives events and faults and deals with them appropriately on behalf of the user process.

Currently, the only use of the unmasked interrupt sources is for Operating System Traps (OSTs). The execution of an OST by a process causes this interrupt to be set, which results in the kernel gaining control of the machine (the kernel services this interrupt source) and providing the requested service. In some instances the service is provided by a kernel process. *UNIX** operating system user-level services are provided by a DMERT supervisor process. In these latter two cases the kernel passes control to the appropriate process via its OST entry.

3.5 Execution levels and priorities

DMERT prioritizes processes into sixteen execution levels (see Fig. 4). Execution level 15 is highest in priority. Kernel processes run at execution levels 2 through 15. Application kernel processes can only use execution levels 3 through 14. Level 2 is reserved for DMERT special processes, and level 15 is used by the DMERT Timer and Error Interrupt Handler. Supervisor and user processes run at execution level 0, and supervisor processes in a critical region run at level 1. Running at level 1 prevents any other supervisor process from interrupting the supervisor process while in its critical region. Within execution level 0, all supervisor and user processes are further prioritized within a 256-level priority structure. Supervisor and user processes are scheduled on a highest-priority basis by the DMERT scheduler. Processes of equal priority are scheduled among themselves using a round-robin scheme.

Kernel processes are dispatched as a result of an interrupt. This

* Trademark of Bell Laboratories.

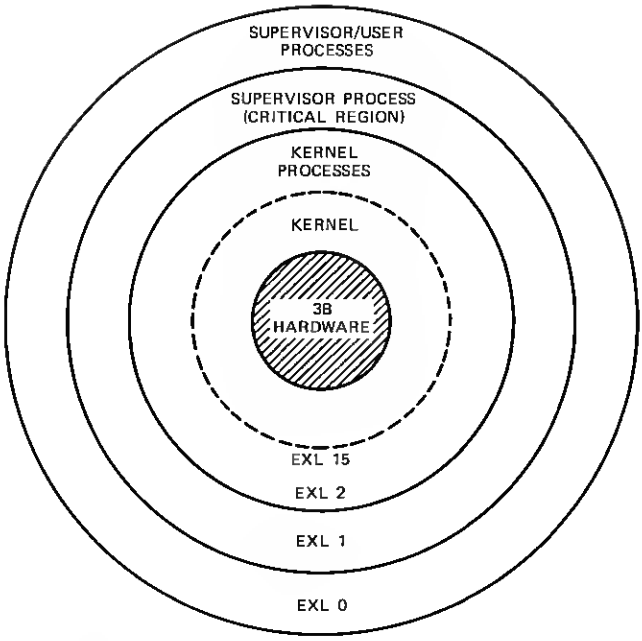

Fig. 4—Hierarchical organization of DMERT.

interrupt may be from a peripheral device or simply a PIR indicating the reception of an event or fault. Each execution level has a unique IM associated with it. The IM for each execution level inhibits all interrupts that are handled at the same or lower execution level. Since supervisor and user processes run at execution levels 0 and 1, they are always preempted by kernel processes.

3.6 Interprocess communication

DMERT provides several mechanisms for interprocess communication. These include events, messages, faults, shared memory, and shared files.

3.6.1 Events

An event is a single bit of information having a predefined and agreed upon meaning between cooperating processes. When an event is sent from one process to another, DMERT will set the PIR interrupt for the level at which the receiving process executes. When that PIR is unmasked (the current execution level specifies an IM that has that PIR unmasked), DMERT will handle the PIR interrupt and dispatch the process for which the event was intended at its event entry. The type of event(s) sent is passed as a parameter.

3.6.2 Messages

A message is a mechanism for transmitting multiple words of data between cooperating processes. The content of the message must be predefined and understood by sender and receiver alike. The reception of a message is indicated to the receiving process by a message event. Except for kernel processes, the contents of a message buffer (the data being passed) is copied from the sender's address space into the message buffer and then to the receiver's address space by various DMERT operations. All kernel processes include the system message buffer segments in their virtual address space. Hence, transmission of messages between kernel processes is much more efficient as the data is actually passed in shared memory.

3.6.3 Faults

Faults are another mechanism for interprocess communication that are usually used to indicate an error, system initialization, or other emergency-type of communication. A fault consists of an 8-bit (byte) fault code that has a predefined meaning between sender and receiver. The reception of a fault causes a process to be dispatched at its fault entry.

3.6.4 Shared memory

Processes can share memory on a segment basis. The entire segment (up to 32K words) would be mapped into the virtual address space of each process. This is the most efficient means of interprocess communication, but it may result in tight coupling between the sharing processes.

3.6.5 Shared Files

Sharing a file is very similar to sharing memory except the storage is done on a secondary storage device (specifically a disk). This mechanism is obviously not as efficient as memory, but provides a media for sharing larger amounts of data.

3.7 DMERT system integrity

3.7.1 Structure and strategy

The DMERT integrity package is based on the duplex, self-checking, nonmatching philosophy of the 3B20D Processor, and the hierarchical organization of the DMERT operating system. It was designed to provide tolerance to both hardware and software faults, so that the 3B20D Processor running under DMERT meets its reliability requirements. The functions provided by the integrity package running under DMERT appear in all abstract machine levels. Their placement in the hierarchical structure depends upon their complexity, the desired real-time response, and the services that they require.

The nondeferrable functions are activated when a hardware or software fault has been detected or a maintenance request has been made via a TTY message. They may initialize one or more system components, reconfigure the system, or just generate status reports. Some nondeferrable functions (e.g., processor initialization) cannot assume operating system sanity and require a fast response time. They are placed in the kernel and are initiated by the hardware self-checking circuits of the processor. Other functions like the recovery from a fault in an on-line peripheral can assume operating system sanity, but they must be performed in the minimum amount of time. Consequently, they are implemented as kernel processes.

Some functions such as diagnostics require services provided by lower abstract machines and their execution can be defferred. These execute under a DMERT supervisor process that provides a *UNIX* operating system environment. The deferrable integrity functions include the initiation and control of the diagnostics, administration of diagnostic requests, and requests to remove or restore a unit to service.

3.7.2 Software integrity

3.7.2.1 Overload and sanity. The focal point of the software integrity package of DMERT is the System Integrity Monitor (SIM). SIM is a kernel process that is responsible for, among other things, system overload control, sanity monitoring, and coordination of initialization actions. SIM coordinates its actions with an application process called the Application Integrity Monitor (AIM). The combined action of these two processes defines the overall software integrity strategy. Further discussion on overload and sanity is presented below as part of the TSPS No. 1B system integrity, which includes a description of the AIM process.

3.7.2.2 Initialization levels. DMERT provides six levels of initialization (levels 0–5). The application can specify sublevels for DMERT levels 0–4. Level 0 is for application initialization only. That is, DMERT does not perform any initialization; DMERT merely notifies the application to initialize. Level 1 results in DMERT initializing and then notifying the application to initialize. In this case, all processing in the machine comes to a halt, the kernel and interrupt stacks are cleared, interrupt sources are disabled, and all kernel processes as well as the currently running supervisor process are faulted. Level 2 is a reboot of the system. At this level, however, certain protected segments of memory specified by the application are not lost, nor is the system Equipment Configuration Data (ECD). A Level-3 bootstrap reloads the ECD from disk, while a Level-4 bootstrap reinitializes all of memory including the protected application segments. Level 4 can only be requested manually. Level 5 is also manually initiated. It

reloads the system disk from tape. A manual bootstrap is then required to initialize the system.

IV. THE SPC 1B

The SPC 1B Processor is the functional entity that replaces the SPC 1A in TSPS No. 1B. In reality, the SPC 1B is an SPC-like environment created on the 3B20D by several components. The Peripheral System Interface (PSI), emulation microcode, 3B20D hardware, native-mode software in the TSPS process, and the DMERT operating system all play a role in realizing the SPC 1B image. This section will describe the basic characteristics of the SPC 1B, briefly discuss its components, and concentrate on those capabilities required to emulate the SPC 1A at the instruction level. Other aspects of the machine and the cooperation of the components are developed in subsequent sections of this paper.

4.1 SPC 1B components

The PSI provides the necessary interface between the 3B20D Central Control (CC) and the TSPS peripheral buses. Emulated programs communicate with the TSPS peripherals without hardware modifications to the peripherals themselves. The microprogrammed control and the flexibility of the 3B20D architecture make it feasible to emulate a machine of vastly different characteristics. Emulated instructions are implemented by a microprogram that co-exists with the microprogram for the native instruction set. An off-line object code post-processor complements the emulation microcode by creating instruction formats optimized for execution on the 3B20D hardware (see Section 4.5).

Finally, some functions must be emulated at the system level. Functions such as unlocking the write protection on an emulated store or requesting a processor switch are handled by native-mode software in the TSPS process. It provides services to emulated software not easily performed in the SPC 1A environment. In many cases, TSPS native-mode code interacts with other processes and the operating system to realize other services. Section 5.1.1.1 discusses TSPS native code in depth.

4.2 Basic characteristics

The characteristics of the SPC 1B are quite different from those of the physical 3B20D. In the 3B20D Processor, all data paths, memory locations, and registers are 32 bits wide. Memory addressing is byte-oriented and 24 bits wide, providing a 16-million-byte capability. Invisible to both software and firmware is the memory management hardware, which provides virtual addressing. A high-speed buffer cache

is also included, which shortens the effective memory access time. The Arithmetic/Logic Unit (ALU) provides ones or twos complement arithmetic by allowing microprogram control of the carry in bit. A Rotate-Mask Unit (RMU) provides right rotates in any amount up to 31. The rotate amount also selects a mask from 16 mask classes to implement shifts and item extraction. In addition to the AND operation with a mask, the OR with the complement of a mask is possible to facilitate operations such as sign extension. There are 16 general-purpose registers in the 3B20D, although three are reserved for stack maintenance.

On the other hand, the SPC 1B, like its predecessor SPC 1A, is a 20-bit word-addressed machine. The arithmetic is done in ones complement and emulates a subtractor circuit. The significance of a subtractor is that the minus zero result is avoided in almost all cases. Seven general registers can be used for indexing, data manipulation, return addresses, and peripheral communication. A null (N) register can also be specified as a source of a zero operand. Rotates and shifts are allowed in both right and left directions. Contiguous bit masks of most sizes from 1 to 20 bits can be used for item manipulation. An insertion masking operation is also available to allow user-specified, non-contiguous bit masks.

4.3 Mapping the SPC 1A image

Since assembly-language programs are designed with an intimate knowledge of the machine they are written for, the 20-bit structure of the SPC 1A is embedded deep into the TSPS software. As a result, operations such as rotation and arithmetic represent a potential source of emulation errors. Because of this and the basic goal of emulating with minimal changes, the 20-bit architecture of the SPC 1A has been retained. Because of the difference in word size between the 3B20D and the SPC 1A and other differences described in the previous section, an image of the SPC 1B must be mapped onto the physical 3B20D hardware.

A 20-bit SPC word is contained right adjusted in a 32-bit 3B20D word. The most significant 12 bits are maintained as zeroes in both registers and memory locations. Twenty-bit word addresses are converted into 24-bit byte addresses by multiplying each address by four and forcing the uppermost two bits to be zeroes. The emulation of 20-bit addressing retains the limitation of 1 million word addressability as on the SPC 1A. Further, the one-million-word emulation address space must start at virtual address zero in the four-million-word TSPS process address space.

Registers zero through seven have been chosen as the SPC 1B registers. This assignment is identical to the numerical encoding on

the SPC 1A, thus simplifying the post-processing of SPC object code. Although the N register is assigned to be register zero, it requires special handling. When used as a source or argument operand, register zero must first be cleared. This guarantees a source of zero in the event that the N register was modified by specifying it as a destination in a previous instruction. The SPC 1B also contains an imaginary ones register, which may be used in register-to-memory and register-to-buffer bus instructions. The 'ones' register of the 3B20D is used for this purpose.

4.4 Instruction implementation

The instruction set of the SPC 1B is very similar, but not identical to, that of the SPC 1A. Maintenance instructions related to the SPC 1A Processor were deleted for the SPC 1B. Other instructions that dealt closely with the SPC 1A hardware have required slight modifications. In spite of these changes, an SPC program or emulation mode programmer cannot and need not distinguish between the SPC 1A and 1B in most cases. Architectural differences described in previous sections are handled by microcode and are invisible to the programmer.

4.4.1 Related 3B20D processor hardware

To provide a framework for the description of instruction implementation, that portion of the 3B20D architecture directly affecting the emulation will be presented. A complete description of the 3B20D hardware can be found in Refs. 4 and 6.

As shown in Fig. 5, the CC is structured around a source and destination bus. The Data Manipulation Unit (DMU) accepts data from the source bus, and gates results to either an internal register or to an external register via the destination bus. The DMU contains the ALU, RMU, general registers, and parity circuits. Another circuit between the buses is the Find Low Zero (FLZ) unit. This circuit accepts 32 bits as input and yields the binary value of the bit position of the least significant zero. Finally, a direct path between the buses exists for fast data transfers between the external registers.

The store interface consists of tbe Store Address Register (SAR), Store Control Register (SCR), Store Data Register (SDR), and Store Instruction Register (SIR). These registers, along with a separate store operation field in each microinstruction, allow store operations to be done in parallel with 3BCC operations. To facilitate instruction fetching, hardware is dedicated to increment the Present Address (PA) register. A Program Status Word (PSW) bit specifies the PA increment amount to be 2 or 4, depending upon whether halfword or fullword mode is desired. The output of the incrementer is loaded into the SAR and back into the PA when a fetch is initiated.

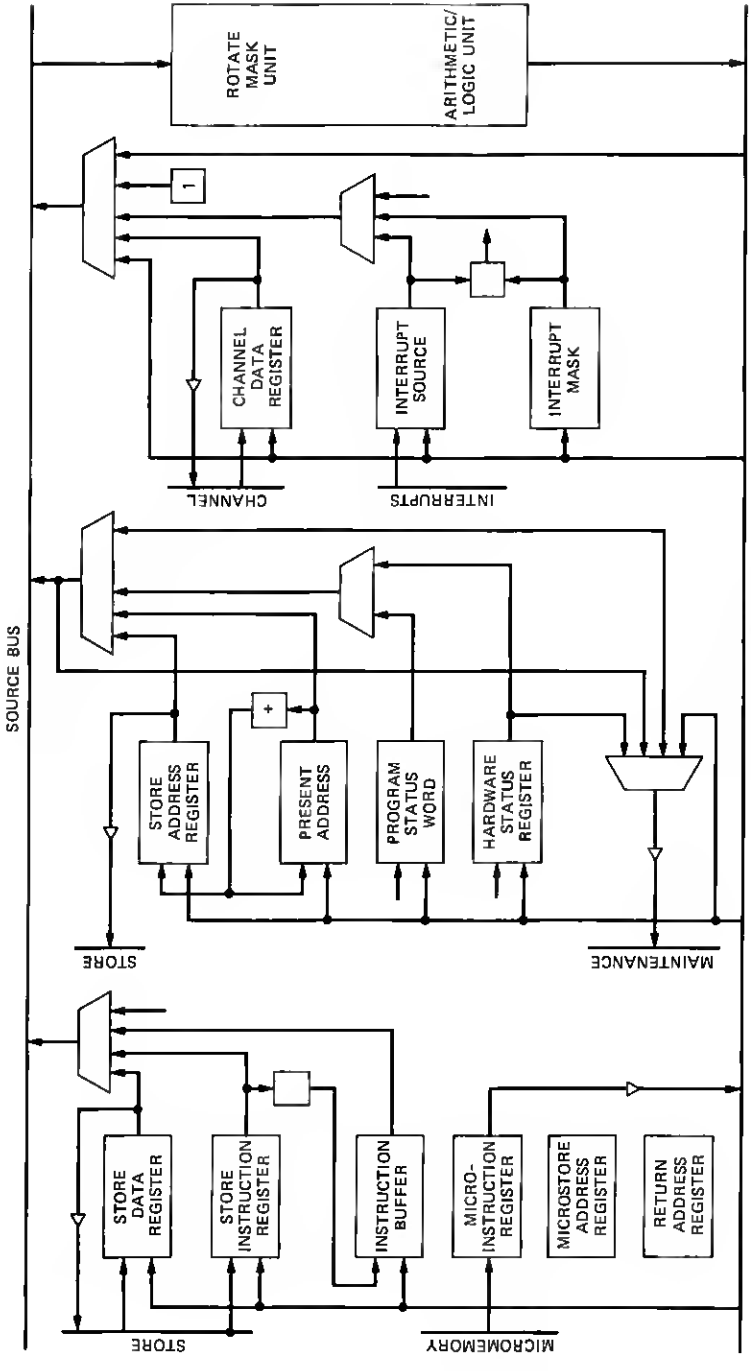

Fig. 5—Processor control architecture.

The basic I/O registers are the Channel Address Register (CAR), and the Channel Data Register (CDR). The Hardware Status Register (HSR) holds status and response information relating to channel operations. Channel operations are performed by microcode via the Pulse Point Register (PPR). Bits in the PPR are set and reset by microcode to form control pulses on the CCIO bus.

4.4.2 Instruction fetching and decoding

An opcode on the 3B20D is eight bits wide. Multiple virtual machines are implemented by providing four complete sets of 256 opcodes. The instruction-initiation operation is a store-field function that operates as follows. Fetched instructions are loaded by the store into the SIR. When a new instruction is to be started, the contents of the SIR are transferred into the Instruction Buffer (IB). The opcode portion and two emulation-mode bits in the PSW are used to form a microstore address, which is the entry point into the microprogram for that instruction. Therefore, the mode bits effectively partition the microstore into four segments and, hence, four instruction sets.

4.4.3 Instruction staging

As mentioned above, the instruction currently being executed is located in the IB. General registers and indices for the 16-way branch microinstruction can be indirectly specified by four-bit fields (nibbles) in the IB. For single-bit testing, the high-order bit of each nibble is available as a condition for the conditional jump microinstruction.

Similarly, there are three fields defined in the IB for rotate and mask operations. For a rotate and mask operation, a microinstruction must specify a mask class, a rotate amount, and masking operation. The rotate amount also selects which mask in the named class is to be used. The three five-bit fields in the IB can be used to specify the rotate amount and corresponding mask to be used.

4.5 Basic operation

The SPC 1B instruction set differs from other sets in that most basic operations can be specified as options on many instructions. Hence, the instruction set is small in number but rich in data-handling provisions. As an example, there is no explicit ADD instruction. Addition can be specified as an option on most instructions. This section will describe the implementation of the basic operations common to many instructions.

4.5.1 Instruction formats

Instructions on the SPC 1A were encoded in a 40-bit double word with a four-bit basic opcode. Opcodes were extended by other bits in

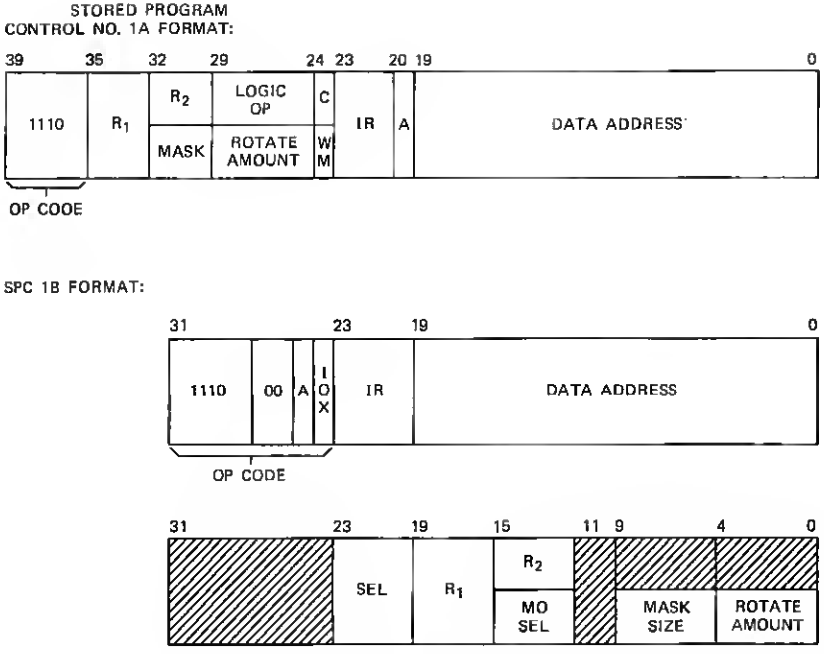

Fig. 6—Memory-to-register instruction.

some instructions, usually in a non-adjacent field. Register fields are three bits. Mask size was specified in a four-bit field that was not strictly binary encoded. The rotate amount is contained in a five-bit field. Options are encoded in multi-function fields within the instruction. In general, these fields were not aligned with the IB fields mentioned above for instruction staging. A bit-for-bit emulation would have resulted in a very inefficient result. Since one of the fundamental goals was an increase in real-time capacity, new formats were designed to provide the most efficient encodings. As an example, Fig. 6 shows the SPC 1A and SPC 1B formats for the Memory-to-Register (MR) instruction.

A post-processor is used to realign the 40-bit SPC 1A object code word into two 32-bit words suitable to execute on the 3B20D. In addition to simple bit shuffling, the post-processor adds information to aid the emulation microcode by performing those computations that can be done off-line. In the same fashion, bits have been included with the basic 4-bit opcode to take full advantage of 8-bit opcode encoding. For example, bits specifying an option can be included in the opcode to eliminate execution time required to decode that option. The unique entry point for each opcode would specify not only the instruction, but

also the occurrence of the option. The post-processor is described in more detail in Ref. 7.

4.5.2 Execution protocol

As mentioned previously, SPC 1B instructions require two 32-bit words to hold the 40 bits of information contained in the SPC 1A instructions. Even in those cases where all relevant information can be held in one word, the spacing of two addresses between instructions must be maintained. As an example, indexed transfers into transfer tables, a common structure in TSPS software, would have had to be recoded had this spacing not been retained. Unlike the SPC 1A, which has a 40-bit memory bus, two separate fetches are required for each instruction. The placement of argument fields in the instruction words has to correspond to the logical execution flow for the instruction. SPC 1B formats are designed to match the flow of execution. Since instructions are a multiple of 32-bit fullwords, the mode bit in the PSW is set for a PA increment amount of 4. In contrast, the 3B20D native instruction set operates on 16-bit halfwords, with a PA increment amount of 2.

The basic fetch-execute protocol is for each instruction to fetch both words of the next instruction. When an instruction is started, the first word is in the IB and the second word is being fetched into the SIR. In this way, the processing of information in the first word can be effectively overlapped with the fetch of the second word. When the first word is no longer needed, the second word is gated from the SIR into either the IB or a scratch register as needed. The SIR is now free to accept the fetch of the first word of the next instruction. The last microinstruction, in addition to requesting that the next opcode be decoded, also starts the fetch of the second word of the next instruction.

4.5.3 Arithmetic

As described previously, the SPC 1B represents negative numbers in ones complement form. A difficulty arises in the end-around carry for a 20-bit word on a 32-bit arithmetic unit. The algorithm used is to insert ones in the upper 12 bits of one of the operands to propagate the carry from bit 19 to the carry-out bit. In addition, a carry-in of one is initially assumed. If the carry-out is a one, then the carry-in assumption was correct and the operation is complete. If a carry was propagated from bit 19 to the carry-out, the upper 12 bits of the result are automatically cleared, as desired. Also, this assumption properly avoids the minus zero result that would have occurred if the carry-in was not made. This is the only case where the carry-in itself forces the carry-out. If the carry-out is a zero, the operation is repeated without the carry-in and with zeroes in the upper 12 bits of both operands. Since

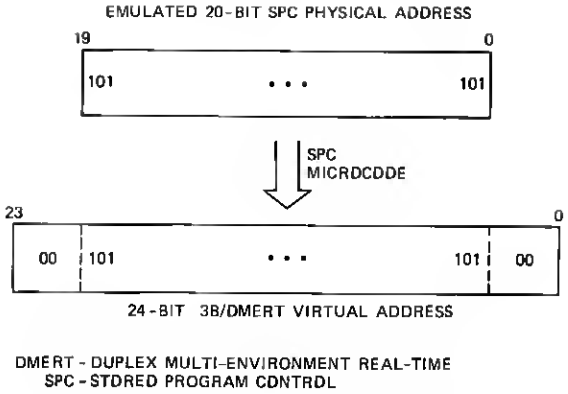

Fig. 7—SPC 1B address translation.

no carry-out of bit 19 will result, the upper 12 bits of the result will be zero.

4.5.4 *Effective address generation*

For those instructions that access memory, indexing of the address is performed by adding the address and the contents of the specified index register. Any of the SPC 1B general registers, including the N register, can be used as an index register. Since negative indices are possible, the indexing operation is done in the same manner as ones complement arithmetic described above. The resulting 20-bit word address is then rotated left two bits (actually left 30 bits) and masked to clear bits 23 to 22 and 1 to 0 to form a 24-bit byte address, as sbown in Fig. 7. Since indexing is costly in terms of execution time, the post-processor detects the specification of N as the index register, and sets an indexing bit to indicate that indexing is not required. The indexing bit is usually contained in the opcode to provide free decoding. The indexing bit, non-existent in the SPC 1A, is an example of information provided by the post-processor to facilitate efficient on-line execution.

4.5.5 *Shifts and rotates*

In the SPC 1B, shifts and rotates are performed over 20-bit data words, even though these words physically reside in 32-bit 3B20D words. Since the 3B20D RMU only accepts right-rotate amounts, all left-direction amounts must be converted by taking their complements with respect to word size (i.e., word size minus left amount equals rigbt amount). For shifts, the proper mask is applied to inject zeroes into vacated bit positions.

Implementation of the wrap-around from bits 0 and 19 for rotates is more difficult. Values less than 12 can be handled by copying the

upper or lower 12 bits of the 20-bit data word into the upper 12 bits of the 32-bit word. For left rotates, the upper 12 bits are used, while the lower 12 bits are copied for right rotates. Rotates greater than 12 can be converted to their equivalent value in the opposite direction before applying the above algorithm. For example, a left rotate of 15 can be effectively realized by a right rotate of 5. It should be noted that rotate and shift amounts specified in the instruction can be converted by the post-processor and encoded as an adjusted value. Rotates and shifts based on the contents of a register must be adjusted during run-time by the microcode.

4.5.6 Data handling

The SPC 1B provides the capability of reading (unpacking) and writing (packing) contiguous data items of almost arbitrary size and position within a 20-bit word. A data item is defined by its size, M, and its position, Q. The unpacking operation involves reading a data word, rotating it right by Q, and applying a mask of size M. The packing operation involves applying a mask and then rotating it into position by the amount Q. This value is then inserted, using a read-modify-write sequence, into the target location.

Since the MQ option is used heavily in TSPS programs, implementation of the rotation in the manner described previously would be very costly. Luckily, in most cases much of this overhead is avoided. When the total of the rotate amount plus the mask size is less than or equal to 20, no wrap-around actually occurs. The more efficient shift operation can be used instead. In TSPS, this is true nearly every time an MQ is specified.

4.5.7 Conditional transfers

The SPC 1B, like the SPC 1A, has no explicit condition flags. However, the results of data manipulations can be tested by conditional transfer instructions. Conditional transfers provide the capability of testing either the entire contents or a specific bit of any general register. The 3B20D condition codes, carry, overflow, zero, and negative, are used for these tests. It should be noted that like the SPC 1A, the minus zero value, coded as all ones, is not detected as zero in these conditionals.

A special conditional transfer instruction is the Detect Right Most One (DRMO) instruction. This instruction transfers if no ones are present in the test register. If a bit is set, the bit position in binary of the least significant one is loaded into a result register. The DRMO and its variation, the DZRMO, which zeroes the rightmost one, are used, for example, by task dispensers to initiate clients based on a job-activity word. The 3B20D FLZ is used to implement these instructions.

4.5.8 Alternate entry points

There are two functions in the SPC 1B instruction set that are unique in that their operation extends across instruction boundaries. They are the Inhibit Interrupt (I) option and the Execute (EXC) instruction. The I option inhibits the J- and H-level interrupts until after the next instruction is completed. Although hardware exists in the 3B20D to unconditionally start tbe next instruction, blocking all interrupts if error interrupts are pending is unacceptable. In this case, the I option is implemented by raising the execution level to block J-level and H-level, starting the next instruction, and restoring the execution level during the instruction following the I-option instruction. The EXC instruction calls for the execution of an instruction at a target address, followed by an automatic return to the next sequential instruction after the EXC. The responsibility of the target instruction is to restore the PA to resume sequential execution.

Since any instruction can follow an I-option instruction or be the target of an EXC, every instruction must determine if a special operation has preceded it. Explicit tests by microcode at the start of each instruction would cause a prohibitive amount of overhead even wben these operations are inactive. Instead, an alternate instruction set is used to force a different entry for the same opcode. The I option and EXC microcode forces the next instruction to be entered at its alternate entry by setting the appropriate emulation-mode bits in the PSW. The alternate entry typically determines which operation to unwind, restores the mode bits to their normal value, and transfers to the normal entry point to execute the instruction. Since the SPC 1B does not have condition codes, the PSW condition flags are available to specify whether the EXC or I option is in effect.

4.5.9 New SPC 1B instructions

There are four new instructions in the SPC 1B instruction set. The Register to Buffer Bus (RBB), Buffer Bus to Register (BBR), and Ones to Buffer Bus (OBB) instructions have been added to reference the buffer bus system. These instructions, along with the buffer bus, are described in Section 5.1.4. The remaining new instruction is the Switch Mode and Transfer (SMT).

The SMT instruction allows emulated programs to transfer to the native mode and begin executing its instruction set. It is patterned after the native-mode CALL instruction to provide a calling sequence consistent with the C language. The SMT puts a stack frame, including two register arguments, on the stack, switches the mode bits in tbe PSW to specify native mode, and transfers to the address specified in the instruction. Two instructions in the native language are also used to support mode switching. They are the Call to Emulation (CALE),

and Return to Emulation (RETE) instructions. Except for the mode switch, these instructions are identical to the standard native CALL and RETN instructions.

The ability to transfer between native and emulated code allows C language to be incorporated into the TSPS process. In fact, the TSPS process has standard C-coded entry points, as described in Section 5.1.1.1. Mode switching eliminates the need for new instructions to perform operations not possible with the emulated instruction set. Examples are Operating System Traps (OSTs), 32-bit data manipulations, and stack accesses. Finally, designers of new features can evaluate on an individual basis which language would be most suitable. Trade-offs can be made between the necessary degree of coupling with existing emulated code and the ease of programming in C.

4.6 Peripheral orders

As mentioned in Section 4.1, the PSI provides the necessary timing and electrical interface to the TSPS periphery. The function of the microcode for peripheral instructions is simply to pass necessary information for the PSI to execute the order. At the end of the sequence, the PSI passes back an answer word and an indication of whether the periphery returned the correct reply signals. A failure indication is used to generate the emulated F-level interrupt, as discussed in Section 5.1.5.1.

The CC communicates with its channels over the Central Control Input/Output (CCIO) bus. An Application Channel Interface (ACHI) is provided with the 3B20D processor to allow the PSI limited access to the CCIO bus. In essence then, the PSI appears as a main channel to the 3B20D processor. Commands and data are sent and data received in a parallel fashion to/from the ACHI-PSI via the CDR CAR, and PPR. Status information, returned by the PSI into the HSR, is used to indicate a failed peripheral order.

The sequencer in the PSI performs four basic functions. In addition to the peripheral order, there are sequences to read a register and write a register. These functions are needed not only for buffer bus references, but also for diagnostic access. The remaining sequence is a pulse sequence, which is used to send special maintenance pulses to the periphery.

All SPC 1A peripheral orders are retained in the SPC 1B instruction set. In addition, a set of PSI instructions have been provided in the native instruction set to execute the four PSI operations. The native-mode PSI instructions were originally designed for the PSI diagnostic, discussed in Section 5.3, which is written in C and native assembly language. They are also used by the Application Integrity Monitor (AIM) process for PSI fault recovery and initialization (Section 5.2.2.1.).

One additional capability provided by the emulation microcode is a set of microcode routines to execute peripheral operations on the off-line processor. These routines accept data from the 3B20D Maintenance Channel (MCH), temporarily release the inhibit on the off-line PSI, and execute the order without making any main-store accesses. By not interfering with the main-store update mechanism, these routines can be executed while the off-line processor remains in the standby mode.

Off-line sequences for pulse and peripheral orders are used by TSPS peripheral fault recognition to retry failed orders and provide better fault resolution. An off-line PSI initialization routine is used by AIM just prior to a soft switch. It is also used periodically by AIM as a hardware audit of the off-line PSI. Should this routine indicate a failure in initializing the PSI, a diagnostic will be requested by AIM. The benefit of this audit is to reduce the latency time for detecting faults in the standby PSI.

4.7 Emulation-dependent software

Although the majority of SPC 1A instructions have been emulated on the SPC 1B, there are functions that do not work in the same manner as on the SPC 1A. In general, the functions requiring modification are in the maintenance programs, which are by necessity more processor-dependent.

The most obvious source of change is the need to recode or eliminate those routines that contain instructions not carried over on the SPC 1B. The most common source of change is recoding buffer bus references to use the new instructions. Unmodified buffer bus references would simply reference a memory location in segment zero since the microcode does not trap the address as a buffer bus address in normal memory-access instructions. A third source of change is due to the new object-code formats and the difference in size between instruction and data words. Since the emulated code can only access 20 bits, instructions cannot be moved or created via data operations. Similarly, data cannot be executed as an instruction because the uppermost 12 bits of data words are zeros. On the SPC 1B, the zero opcode is interpreted as the ANOP instruction. This protective measure would trap a wild transfer into a data area by generating an A-level interrupt. Finally, usage of part of an instruction as data may not work because of the shifting of fields in the new object-code formats. An example of this in TSPS No. 1 code is to read the address field of a transfer instruction and store it to be used as an address later.

A more subtle difference is the execution time of instructions on the SPC 1B. The SPC 1A is a fixed-cycle machine, and instructions are made up of a number of these basic cycles. The execution time of an

instruction could be determined exactly, independent of the options specified. A common dependence on the execution time of an instruction is in timing loops. A precise time delay is created by executing a loop the correct number of times.

Since the 3B20D executes emulated instructions several times faster than the SPC 1A,[8] timing constants coded for the SPC 1A produce delays proportionately smaller on the SPC 1B. The modification of these constants is not quite as simple as multiplying by a speedup factor for the following reasons. First, the SPC 1B instruction set is microprogrammed, and thus every option must be accounted for when calculating the execution time for an instruction. Second, since the 3B20D employs virtual addressing, memory management delays owing to translation are incurred as a function of program flow. The cache memory also plays a role by shortening the memory access time when the access is contained in the cache. Since the cache is shared by all processes in the system, its effect is even less predictable. Finally, Direct Memory Access (DMA) activity steals memory cycles from the processor and represents an invisible form of interference.

As a result, timing loops with tight window tolerances cannot be guaranteed. In several cases, routines that performed window timing required recoding. On the other hand, minimum timing poses no problem. Best-case estimates can be made to determine the minimum execution time for a program segment by assuming no translation or DMA delays and all cache hits. Any delays actually incurred during execution serve only to lengthen the program segment time, which is acceptable for minimum timing.

V. TSPS NO. 1B SOFTWARE

This section describes how the TSPS software was ported to run under the DMERT operating system. In addition, it describes the system integrity software and other processes required to complete the emulation.

5.1 The TSPS kernel process

The emulated TSPS software has been incorporated into a single, large, nonkillable kernel process under DMERT. This is due largely to the real-time constraints of TSPS software operation and its existing structure on the SPC 1A. All TSPS code on the SPC 1A shares a single physical address space. All data and programs are equally accessible from any other program. The entire software structure is very tightly coupled. Subdividing the software into several processes would have required a restructure and redesign of the TSPS software. Also, the operational timing constraints of TSPS software (e.g., 5-ms I/O processing, interject responsiveness and base-level E-E times) require

TSPS to be memory-resident and interrupt-driven at the kernel-process level.

5.1.1 Emulation environment

Emulation of existing TSPS software on the 3B20D Processor required the creation of an SPC 1A environment. That is, in order to work properly the TSPS software had to be shielded from both the actual physical machine (3B20D) it was running on and the DMERT operating system it runs under. This environment has been established with a combination of hardware (PSI), firmware (the microcoded SPC 1A instruction set), and software (native-mode code within the TSPS process).

5.1.1.1 Native-mode code. Except for the emulated code within the TSPS process, all software in the TSPS No. 1B system is coded in 3B20D native mode. Most of the software is coded with the C language. Hence, the 20-bit emulated code exists in a 32-bit universe.

Figure 8 illustrates the structure of the TSPS process. The TSPS process is dispatched (given control of the machine by DMERT for execution) at either its interrupt, event, or fault entry. These entry points are coded in 3B20D native mode. Thus, the TSPS process always begins execution in 3B20D native-mode code. These entry points determine what type of processing is to be performed and transfer, while simultaneously changing instruction sets, to appropriate routines within emulated software.

In addition to the native-mode entry points, the TSPS process also contains native-mode routines for communication with DMERT and other processes and for performing certain functions which the emulated code is incapable of doing. As a result, the native-mode code within the TSPS process completely isolates the emulated code from the 32-bit universe surrounding it. Hence, to the rest of the system the TSPS process appears to be a typical (albeit very large) native-mode kernel process.

5.1.1.1.1 Interrupt entry. The only hardware interrupt source to which the TSPS process is attached is a clock-driven 5-ms interrupt for I/O processing. This single interrupt source is used for both J- and H-level processing. This same 5-ms clock pulse is transmitted to the PSI to synchronize its clock with the processor. The PSI clock transmits clock pulses to TSPS peripherals. When the interrupt source fires, the TSPS process is dispatched at its interrupt entry point. The interrupt entry is coded in 3B20D native assembly language because of its simplicity and need for efficiency: it runs every 5 ms. Its only functions are to determine whether this execution is a J- or an H-level, set the appropriate bit in the interrupt-level activity flag buffer bus word, and transfer to the appropriate emulated routine.

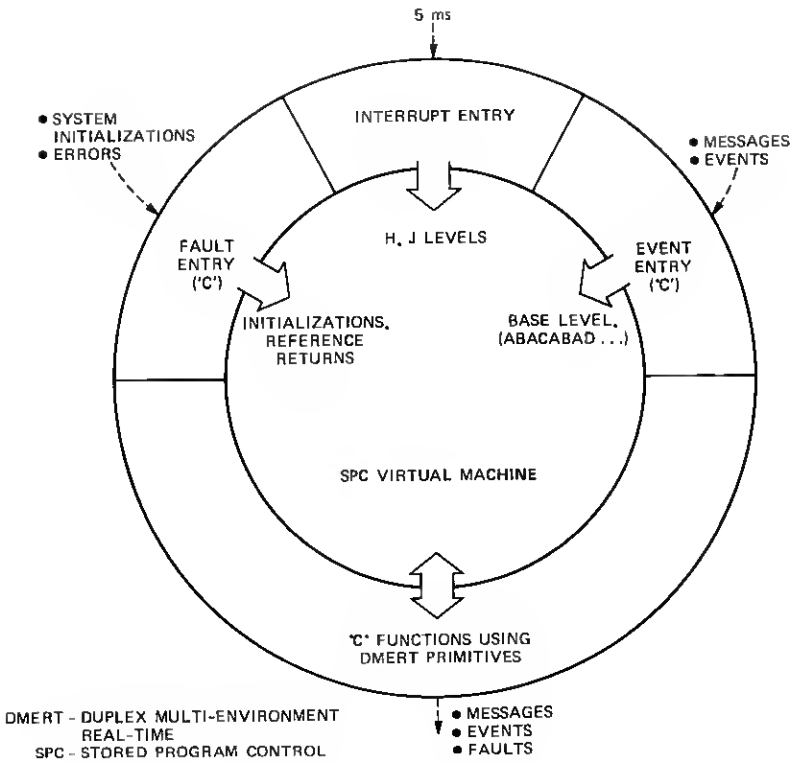

Fig. 8—TSPS Process.

Handling both J- and H-level interrupts with a single interrupt source is accomplished by using two execution levels for J-level processing. When TSPS is dispatched at the interrupt entry to handle a J-level interrupt, it runs at execution level 12. While at level 12 the interrupt source is masked off by the level-12 interrupt mask. Hence, the interrupt entry cannot be re-entered. When high-priority J-level work is completed, the H-level inhibit bit in the buffer bus is cleared. In doing this the microcode for the buffer bus write operation drops to execution level 11 for low-priority J-level work. This is possible since no other processes in the system run at execution levels 11 or 12. At level 11, the 5-ms interrupt source is unmasked and will allow the TSPS interrupt entry to be re-entered if J-level does not complete within 5 ms. The interrupt entry reads the buffer bus to determine whether a particular execution is a J- or H-level. If neither activity bit is set, then it is a J-level and the J-level activity bit is set. If the J-level activity bit is already set, then an H-level has occurred and the H-level bit is set. In the latter case low-priority J-level work has been interrupted and its state has been saved on the interrupt stack.

Although the interrupt handling begins in the native-mode interrupt entry, the return from interrupt is always performed from emulated code. At the end of H-level, the last job executes an emulated EGBN instruction. The EGBN clears the H-level activity flag in the buffer bus and performs a return from interrupt sequence, which causes the saved J-level state to be popped from the interrupt stack. J-level then resumes execution at the interrupted point. At the completion of low-priority J-level an emulated GBNHJ instruction is executed. The GBNHJ clears the J-level activity flag, sets the H-level inhibit bit, and does a return from interrupt. Hence, the process that was executing when J-level began is resumed at the point of interrupt. This interrupted process itself may be the TSPS process performing base-level work.

5.1.1.1.2 Event entry. The TSPS process event entry is coded with the C language and runs at execution level 5. An event sent to the TSPS process causes the process to be dispatched and begin execution at its event entry. The primary events sent to the TSPS process are for initialization upon creation, interprocess message reception, and time-outs requested for real-time breaks in TSPS-base level processing.

5.1.1.1.2.1. Initialization event. An initialization event is sent to a process upon its creation. As TSPS is a nonkillable kernel process, this will occur only after a system bootstrap. The native-mode code initializes the process (i.e., attaches to the 5-ms interrupt source, connects to a system port for message reception, etc.) and prepares to take an SIA or SIB call-processing recovery phase.

An important function that is performed during the initialization is setting the proper page protection over the emulated-code address space. DMERT creates processes with protection set on segment boundaries. The TSPS process, however, emulated the SPC 1A protection mechanism. Hence, some segments may have both protected and unprotected areas. The protection within these segments must be modified by changing the protection on the appropriate pages. TSPS uses a DMERT OST for this run-time page-protection change. This same OST is used by the emulated recent change programs when applying a modification to protected memory.

5.1.1.1.2.2 Time-out event. The TSPS process is only one of many processes time-sharing the SPC 1B. The TSPS process must voluntarily take frequent real-time breaks to allow other processes at execution level 5 and below to execute. (Base-level processing is done at execution level 5, which is the lowest execution level at which the TSPS process runs. This is described in more detail below.) These real-time breaks are performed by issuing a time-out request to the DMERT timer* and then relinquishing control of the machine. (The DMERT timer is

part of the kernel that executes at execution level 15. It runs every 10 ms as a result of clock-driven interrupt source. The Timer maintains the DMERT system clock and provides general-purpose timing for all DMERT processes.) At the end of the time-out period the timer will send TSPS a time-out event. When TSPS takes the break, it remembers where in emulated code it has left off. Upon receipt of the time-out event it will resume execution within emulated code at the same place where it had left off. A related event is one sent by the scheduler when the system is idle to wake TSPS up early, before the time-out event has fired. This is described further in Section. 5.1.6.

Currently there are five areas where the TSPS process takes real-time breaks. These are during memory zeroing in an SIA or SIB, during initialization timing loops that are longer than 10 ms, during audit stitching, at the end of an SIA or SIB while waiting for the J-level portion of the phase to complete, and during the base-level E-E cycle.

5.1.1.1.2.3 Message event. The TSPS process receives interprocess messages from other processes. TSPS is notified of the message reception with a message event. When TSPS fields the event, it will process all queued messages and take whatever actions are appropriate. Some uses of interprocess messages by the TSPS process are discussed in the later section describing other TSPS application processes.

5.1.1.1.3 Fault entry. The TSPS process fault entry is also coded in the C language. The TSPS process is dispatched at its fault entry as a result of a processing error (i.e., a protection violation or an invalid address), a system initialization, or a fault sent by another process as a means of interprocess communication. The TSPS fault entry runs at execution level 12 so that recovery actions can be taken without interference from J-level processing.

5.1.1.1.4 Additional routines. Besides the standard process entry points there are other special-purpose native-mode routines included within the TSPS process. These routines are used to interface with DMERT (e.g., to execute an OST), to use interprocess communication mechanisms such as events and messages in order to communicate with other processes, and to implement some functions that cannot be performed in emulated code. In addition, some new code added to the TSPS process was coded using the C language for development convenience.

5.1.2 Address space

TSPS is currently designed as a DMERT "small" process. A small process can have up to 64 segments, and, hence, a virtual address space as large as two million words. Figure 9 shows the TSPS process virtual address space as a small process under DMERT.

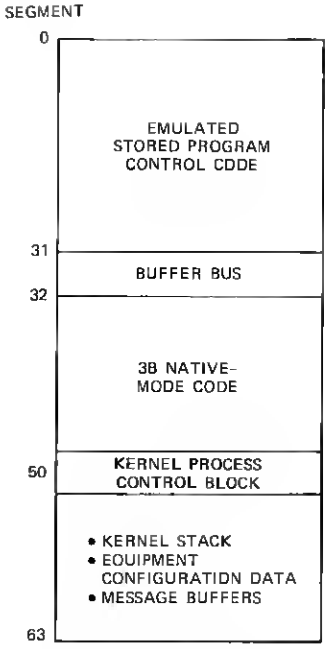

Fig. 9—Virtual address space in the TSPS process.

5.1.2.1 Emulated code. The emulated code resides in the first 32 (0–31) segments of the virtual address space. Thus, the emulated code resides in the first one million words of the virtual address space. This, in fact, is the only part of the virtual address space that is directly addressable by the emulated code. As mentioned previously, all addresses in the emulated code are 20-bit word addresses as in the SPC 1A. A 20-bit address can only address one million words. Hence, the 20-bit physical word addresses on the SPC 1A are now 20-bit virtual addresses in the TSPS process on the SPC 1B.

All 32 segments are fully allocated to their maximum size of 32K words. In effect each segment emulates an SPC 1A store name code. The emulated address space, however, includes segments 0 and 31, whereas store name codes 0 and 31 do not exist on the SPC 1A. Emulated buffer bus references are mapped by the microcode into segment 32. As a result, the emulated code within the TSPS process has a full 1M word address space.

5.1.2.2 Native-mode code. The remaining 31 segments are used for native-mode code within the TSPS process. Not all of these segments are currently used. Those that are used contain TSPS code and data, the process control block, and shared DMERT segments such as the ECD, the system message buffers, and the kernel stack.

5.1.3 Native-mode/emulated-code interface

The TSPS process contains both emulated SPC 1A and native-mode code. As all processing begins in native code, all emulation-code routines are called as functions (subroutines) from the process entry points. Similarly, native-mode routines can be called as subroutines from the emulated code.

All processes under DMERT use a program stack to save the state of a function when it calls another and to pass arguments. Kernel processes (including TSPS) share a common kernel stack with the kernel. A standard stack frame must be maintained when calling another function. Similarly, a standard return sequence is followed. To maintain sanity while transferring between emulated and native code, the TSPS process follows the same protocol. The mechanism used by the TSPS process was patterned after the structure used to transfer between native-mode assembly code and C-language code.

5.1.4 Buffer bus

As described in Section 2.2, the buffer bus on the SPC 1A is a software-accessible set of hardware control and status registers. Since the microcode emulates the hardware action on the SPC 1B, accessing one of these registers causes the microcode to perform a special action, emulating what the hardware would have done. Hence, firmware must be aware that a buffer bus location is being accessed, and what action is required. To eliminate the overhead required to test each memory address in memory reference instructions, new instructions have been added to explicitly reference the buffer bus.

The Register to Buffer Bus (RBB), Buffer Bus to Register (BBR), and Ones to Buffer Bus (OBB) instructions are the exact images of their SPC 1A counterparts, RM, MR, and OM. Converting an SPC 1A buffer bus reference requires only a change to the instruction mnemonic. The address and option fields are identical. The addition of these instructions has the added benefit of not having to set aside store zero as buffer bus images, thus allowing an additional 32K words of usable memory. Depending on their function, buffer bus registers are mapped into either memory locations outside of the SPC 1B address space, PSI internal registers, or 3B20D hardware registers.

Not all registers have been carried over from the SPC 1A. Locations used only in processor and store maintenance were removed along with their corresponding programs. Other locations have been added to maintain and access the PSI. Still others are partially retained, with individual bits having been removed. The set can be broken down into three components: interrupt system, craft interface, and peripheral system.

Locations related to the interrupt system are the Interrupt-Level

Activity Flags (ILAF), Maintenance Interrupt Sources (MAIS), Interrupt Inhibits (PEST), 3B20D Inhibits (3BPEST), and the Millisecond Clock State (MSEC). With the exception of the MSEC register, these are discussed in Section 5.1.5. MSEC on the SPC 1B is the value of the 3B20D Interrupt Timer. The Interrupt Timer is used as the 5-ms clock source for the TSPS J-level interrupt. Access to this timer allows programs to predict when the next J-level will occur. This function is necessary for clients such as diagnostics, which must synchronize their actions with J-level.

Buffer bus locations related to the craft interface are Maintenance Control Center Data (MCCD), MCC Interrupts (MCCI), and Emergency Recovery Display (ERD). The hardware of the SPC 1A MCC has been replaced by the 3B20D craft interface hardware, and DMERT and TSPS craft software. These locations reside in memory locations accessible by TSPS native-mode craft software. The craft interface is discussed in more detail in Section 5.3.3.1.

The remaining locations are all related to the peripheral system. The PSI hardware directly implements many of these locations. Locations emulated by the PSI hardware require that microcode send the necessary command to the PSI to access data or perform some hardware action.

5.1.5 Interrupt structure

5.1.5.1 Emulated interrupts. The emulated interrupt structure consists of A-, E-, F-, H-, and J-level interrupts. Of these, only the 5-ms H- and J-level interrupts are driven by a 3B20D hardware-interrupt source. A-level interrupts are either emulated by native-mode software or generated by the microcode as the result of executing an illegal opcode or an ANOP instruction. E-levels are emulated by native code as the result of fault codes received from DMERT, and F-levels are generated either by microcode or native-mode code.

The microcode for the ANOP instruction traps to an illegal instruction routine in the native microprogram that generates an error interrupt. The DMERT kernel handles the error interrupt, and immediately faults the TSPS process. The TSPS process fault entry determines that an ANOP instruction was the cause of the fault, sets the A-level activity flag in the buffer bus and transfers control to the emulated A-level interrupt-handling routine.

An A-level interrupt can be simulated by native software through setting the A-level activity bit in the buffer bus, which raises the execution level to 12 (the level at which A-level runs), filling in the interrupt-state bin locations, and transferring to the A-level interrupt routine. This is done, for example, to indicate a manual request for a recovery phase. The TSPS process fault handler will emulate the SPC

1A hardware by "generating" an A-level interrupt in the aforementioned way. When the A-level routine is entered, it appears to the software that a hardware interrupt had occurred.

SPC 1A store-error E-level interrupts are not emulated, as this type of error is not seen by the TSPS process. They are handled completely by DMERT fault recovery. Software E-levels (i.e., protection violations and bad addresses) are fielded by TSPS as faults. When this type of error occurs, the TSPS process is immediately interrupted and entered at its fault entry with the state of the machine at the time of the error passed as parameters. The fault entry sets the E-level activity bit in the buffer bus and transfers to emulated code to handle an E-level interrupt. Again, to the emulated code it appears just as if a hardware interrupt had occurred on the SPC 1A.

During the execution of an emulated I/O instruction or buffer-bus instruction,* any failures will be detected by the microcode. The microcode 'generates' an F-level interrupt by setting the F-level activity bit in the buffer bus, saves the address of the interrupted program in the F-level bin, and transfers directly to the emulated F-level routine.

Most PSI errors are detected by the microcode also. An F-level is also generated in these cases. This is done so that F-level can perform an on-line and, if necessary, an off-line retry before letting AIM take PSI recovery actions. In the few cases where a PSI error interrupts AIM first, AIM will fault the TSPS process. The fault entry will then emulate an F-level in the same way as E-levels are handled. There is another class of errors that generate 3B20D error interrupts. The DMERT Error Interrupt Handler (EIH) will field these and fault the running process. In these cases TSPS passes control (via a fault) to AIM for recovery actions.

The microcode and native code must of course be aware of interrupt priorities. If TSPS is already in A- or E-level processing, then the microcode must only set the F-level interrupt source bit and continue processing. An F-level will be generated by the microcode if the source is still set when an EGBN is performed from A- or E-level. Also, if the F-level inhibit is set in the buffer bus the error must be ignored. Setting the J-level inhibit while in base level causes the microcode to raise the execution level of the process to 12. The interrupt mask for execution level 12 masks off the 5-ms interrupt source. Similarly, the "I" option is implemented via the microcode. The microcode guarantees that both J- and H-level interrupts are inhibited for the instruction with the I option and the next instruction executed.

---

* As some of the buffer-bus registers are contained in the PSI, a failure while trying to access these registers is considered a PSI error.

5.1.5.2 Returning from interrupts. The emulated EGBN instruction works the same as it does on the SPC 1A for A-, E-, and F-level interrupts. Because H- and J-level interrupts are driven by a hardware-interrupt source, the EGBN for H-level and the GBNHJ instruction used by J-level operate differently. This difference (described below) is undetectable by the programmer.

When a 5-ms interrupt occurs, the state of the interrupted process is saved on the 3B20D interrupt stack. The return from interrupt (EGBN for H-level or GBNHJ for J-level) must restore tbe state from the interrupt stack and not from memory-resident bin locations. Hence, these two instructions perform the same function as on the SPC 1A, but the actions taken to restore the interrupted state of the machine are different.

5.1.6 Emulated program structure

The emulated program structure within the TSPS process is almost identical to that which exists on the SPC 1A; the major difference is that processing always begins in native code. The native code sets up the proper environment and passes control to the emulated software for the bulk of the processing.

H- and J-levels still run under control of ECIO. As pointed out earlier, the only difference is that a single interrupt source is used on the SPC 1B. The front-end native code determines whether an H- or J-level has occurred by the state of the buffer bus. Different execution levels are used to unmask the H-level interrupt source when going from high- to low-priority J-level.

Base-level programs still run under control of ECMP. Base level is essentially called as a large subroutine of the event handler. Rather than run continuously, as on the SPC 1A, base level must voluntarily give up control of the machine to allow lower-priority processes (processes that run at execution levels below 5) to run. Base level requests a time-out event from DMERT when at least 10 ms have elapsed, and then exits.

There are two ways for base level to be re-entered in response to its time-out request. If during the real-time break there are no processes ready to run, the DMERT scbeduler will enter its idle loop at the lowest system priority. At that point, the scheduler finds that the TSPS process has requested, via an OST during its initialization event entry, that a specific event be sent by the scheduler whenever the idle loop is entered. The event sent by the scheduler causes TSPS to be entered at its event entry, thus waking base level up early. If the idle loop is never entered during the break, base level must wait until the full duration of the time-out request has expired. In this case, receiving

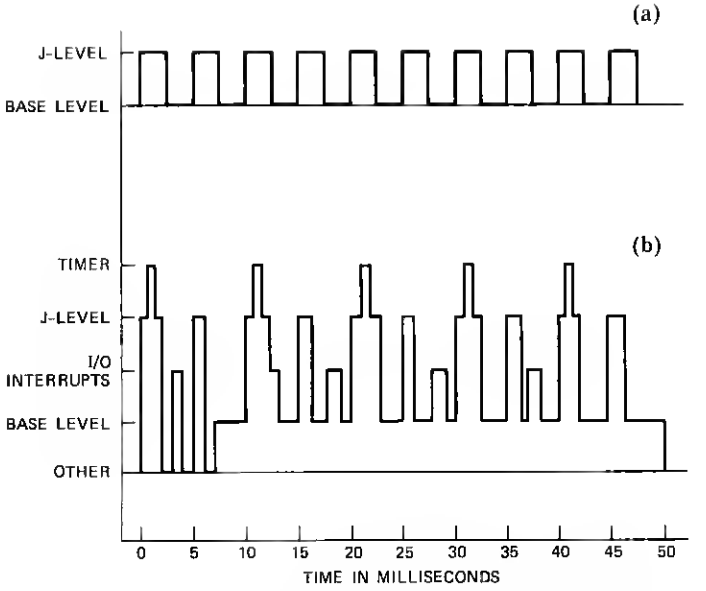

Fig. 10—Interaction between processing levels of (a) SPC 1A and (b) SPC 1B.

the time-out event from the DMERT timer causes base level to resume execution.

This early wakeup mechanism employed by TSPS has a significant impact on the real-time profile of the system. By eliminating idle system real time, base level executes more E-E cycles. This is especially true when the offered call load is light and real time is ample. A higher E-E rate improves the execution of TSPS diagnostics, audits, and other services that are run as a function of E-E rate. The full impact of the early wakeup mechanism is described in Ref. 8.

Base-level code in ECMP was modified to take its real-time breaks between priority classes. There are five breaks per E-E cycle: one after each C and E priority class. Figure 10 contrasts the interaction between various processing levels on the SPC 1A versus the SPC 1B. On the SPC 1A, base level runs continuously, only being interrupted (in the absence of errors or manual actions from the MCC) every 5 ms by J-level. On the SPC 1B, however, there are other processes executing concurrently with TSPS. DMERT I/O processes run at execution levels higher than base level, but lower than J-level. J-level executes every 5 ms whether or not base level is executing or taking a real-time break. Table II shows the execution levels of various DMERT processes, the TSPS process, and AIM.

The real-time breaks are not noticeable to the emulated software. They merely appear as an additional priority class placed in between

Table II—DMERT execution levels

Level	Processes
15	Error Interrupt Handler, Timer
14	Generic utilities
13	System and Application Integrity Monitors
12	TSPS (Fault entry, A, E, F, H and high-priority J)
11	TSPS (low-priority J)
10	Disk and IOP drivers
9	—
8	—
7	File Manager
6	—
5	TSPS (Base level)
4	—
3	Plant measurements, Field update
2	Special processes (Scheduler and Memory Manager)
1	Supervisor processes (critical region)
0	Supervisor/User processes

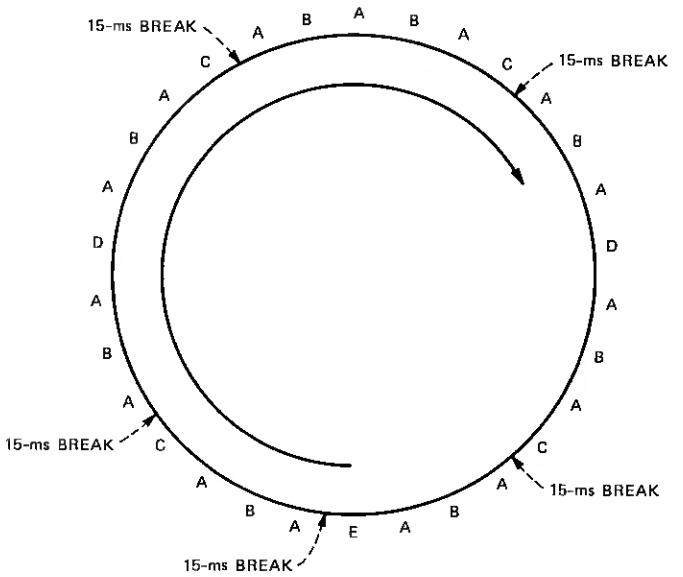

Fig. 11—TSPS No. 1B base-level loop.

the two priority classes separated by the break. Figure 11 shows the modified base-level loop with the five real-time breaks. The insertion of these breaks, though, has required one minor change to interject processing. To guarantee proper interject responsiveness additional interject checks have been inserted. A cbeck for interject work is made after each real-time break. Hence, interject is serviced within 20 ms as a result of a base-level real-time break. Once the interject check is made, base-level processing resumes with the next scheduled priority

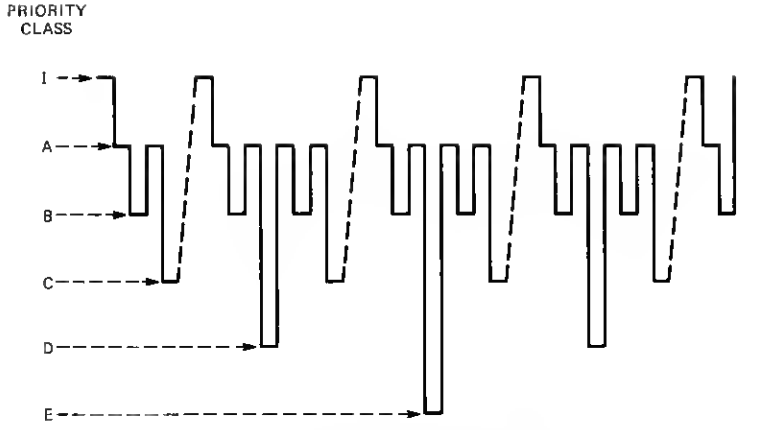

Fig. 12—TSPS No. 1B base-level, priority-class frequency.

class. Figure 12 shows the base-level priority-class execution with the added interject checks.

Another important point is the relationship between the DMERT timer and J-level execution. As the timer runs at execution level 15, it will interrupt J-level. The 10-ms timer interrupt, however, is offset from the 5-ms interrupt by 1 ms. Hence, the timer runs every 10 ms, 1 ms after J-level begins. This guarantees proper operation of the Automatic Message Accounting (AMA) data recording. The AMA data transfer routine must write data to be recorded to the AMA within 1 ms after the 5-ms interrupt occurs. (Recall that the AMA tape unit is synchronized with the 5-ms clock pulse.) Since the timer will not run until this 1-ms window has passed, it does not interfere.

5.2 System integrity

The TSPS No. 1B integrity software consists of three distinct software packages. In addition to the software package provided with DMERT and the emulated TSPS No. 1 integrity package, a new integrity software package was developed. The major functions of this software are:

(*i*) Interface and coordination of integrity-related activities between DMERT and the TSPS application (e.g., initialization, overload, and processor switch)

(*ii*) Sanity and integrity of application processes

(*iii*) PSI maintenance.

These functions have been implemented in the Application Integrity Monitor (AIM), the native-mode portion of the TSPS process, and the PSI diagnostic process.

5.2.1 Hardware integrity

The TSPS No. 1B hardware architecture and, in particular, the design of the PSI, made it possible to retain most of the peripherals used in TSPS No. 1. As a result, the maintenance strategy for these peripherals remained virtually unchanged and the maintenance software implementing it was emulated with only minor changes. The entire maintenance software for the TSPS peripherals resides in the kernel TSPS process and represents a distinct maintenance package.

Unlike the SPC 1A, the SPC 1B does not provide matching between the on-line and off-line processors. Instead, both the 3B20D and the PSI employ extensive self-checking hardware circuits to detect most service-affecting faults. The basic switchable entity is the simplex SPC 1B. When a fault is detected in the on-line processor, a switch to the off-line processor is performed. This switch may be followed by an initialization sequence. The faulty circuit pack is then identified with the aid of a diagnostic. The SPC 1B hardware is designed such that no single fault in either processor can cause a system outage. The TSPS No. 1B hardware architecture, including the PSI, and its maintenance, is described in Ref. 6. TSPS No. 1B reliability is covered in Ref. 8.

5.2.2 Software integrity

5.2.2.1 Application integrity monitor. The Application Integrity Monitor (AIM) process is the sole application interface to DMERT for system integrity issues—initialization, sanity, processor switches, and DMERT overload conditions. Primary responsibility for monitoring the sanity of the TSPS application and controlling its recovery resides with AIM. In addition, AIM performs PSI fault recovery and initialization. Reports of DMERT recovery actions and resource overload are made to AIM, which in turn controls the application response to the event.

AIM is a nonkillable kernel process running at the same execution level as the DMERT System Integrity Monitor (SIM), higher than all other application processes. It interfaces with SIM via interprocess messages. AIM executes briefly every 500 ms at execution level 13 or as a result of PSI error interrupt or faults sent from the TSPS process. Since AIM is at a higher execution level than the TSPS process, it runs immediately when faulted by TSPS. This is done, for example, when TSPS requests a phase. TSPS faults AIM with its request, and AIM requests the phase of DMERT and reports back to TSPS about when to start its initialization.

5.2.2.2 Initialization. The TSPS call-processing recovery phases were integrated into the DMERT initialization levels to provide a coherent strategy. In the resulting initialization mechanism, between two and four TSPS initialization levels are associated with each DMERT

initialization level. Table III shows the mapping of TSPS recovery phases to DMERT initialization levels. TSPS minor and major phases are not taken after a bootstrap (DMERT initialization levels 2 through 4). The bootstrap recreates the TSPS process in memory, zeroing all its unprotected data areas. Hence, a minor or major phase will not initialize the TSPS process; an SIA or an SIB is required.

The TSPS No. 1B initialization mechanism employs a sequential escalation for most transitions. In addition, TSPS executes four SIB phases, with each using a different peripheral configuration, before escalating to the next higher DMERT level. However, in some cases, such as manual initialization or an interrupted initialization, the sequential escalation may be bypassed. Control of the escalation sequence is distributed between SIM and AIM. If the highest automatic initialization phase [DMERT level 3 and TSPS level 4 (SIB)] is reached and the system still does not recover, the system will loop continuously, rebooting the system and taking SIBs with varying peripheral configurations until the system recovers or manual action is taken.

A few minor changes were made to the initialization code to allow the TSPS process to take real-time breaks during recovery phases. Since ECMP is not cycling through priority classes during phases, it was necessary to add other breaks during the phases. Continuous running of TSPS during a several-minute-long phase would cause several DMERT sanity and overload checks to fail as lower-priority processes would not be run. Thus, during a phase the TSPS process will break during memory zeroing, for timing loops greater than 10 ms in duration, during audit stitching, and while base level waits for J-level to signal the end of initialization. The length of the phase is not affected by these real-time breaks. The length of the phase is determined by the amount of peripheral equipment, and hence the number of 5-ms interrupts (J-levels) that must occur for peripheral equipment initialization. Although each interrupt will execute faster, the total

Table III—DMERT level (EAI command)

	0 (Application Only) (50)	1 (51)	2, 3, 4 (Bootstrap) (52, 53, 54)
Application parameter			
0	No action	Reference return	Boot without TSPS process
1	MNA	MNA	Boot + SIA
2	MJA	MJA	Boot + SIA
3	SIA	SIA	Boot + SIA
4	SIB	SIB	Boot + SIB
L	Limp mode	Limp mode	Boot + limp mode
Other values	No action	Reference return	Boot + SIA
No value (null)	No action	Reference return	Boot + SIA

elapsed time depends on the number of interrupts and not the speed at which they execute.

5.2.2.3 Reference returns. Taking reference returns in the TSPS process is more difficult than on the SPC 1A. All function calls, whether they be between native-code routines or between native code and emulated code (or vice versa), maintain a history of the function call on the process stack. (Transfers between emulated routines do not affect the stack. These transfers work just as on the SPC 1A. In fact, emulated code cannot explicitly access the stack and is unaware of its existence. Only the microcode for the SMT instruction manipulates the stack to maintain a proper stack frame when calling a native routine.) A function must make a normal return to the calling routine to properly unwind the stack. Direct transfers between functions would quickly result in an incongruous stack from which the process would have no way of properly returning to the calling routines. Hence, when taking a reference return, the TSPS process must properly unwind the stack.

To further explain reference returns within the TSPS process, consider a fault generated as the result of an invalid address. The TSPS process will be interrupted immediately, entered at its fault entry, and passed the state of the machine at the time the fault occurred. The fault entry then "generates" an E-level interrupt. The state information indicates the instruction in error and the contents of all the general-purpose registers at the time of the error. Because the TSPS process contains both native and emulated code, this error could have occurred in either. In addition the error could have occurred at a point many levels deep in nested function calls. If a direct transfer to a known "safe" reference point is performed, then a later return to the original calling function (i.e., the event or interrupt entry) will fail as the program stack will not have been unwound properly. To accommodate the existence of the stack, slight modifications to the handling of reference returns had to be made.

If the E-level occurs in H- or J-level, an immediate return from interrupt (using an EGBN or GBNHJ) will be executed. The return from interrupt will clear all of the interrupt's function call entries from the stack and allow the interrupted process to resume processing. This has the effect of canceling not only the current J-level job in progress (as on the SPC 1A), but also all jobs scheduled to execute that 5-ms entry.

If the E-level occurs in base level, a similar situation exists with respect to the kernel stack. In this case, however, a simple return from interrupt will not suffice. Base level must stimulate itself to be re-entered at some later point in time. (J-level, of course, is re-entered by the next 5-ms interrupt.) This is normally done with a time-out request.

This is not sufficient here, either, as base level must know that its next entry is for the purpose of taking a reference return rather than continuing its base-level loop. Hence, base level sends a fault to itself and then performs the return from interrupt. As before, the return from interrupt will clear the stack of any function call entries made by base-level processing. When the fault entry is entered, the fault code will indicate that a base-level reference return is to be made. The fault entry drops to execution level 5 and transfers to the base-level reference point.

The base-level reference-return handling is functionally equivalent to that on the SPC 1A. The only difference is that a momentary delay is experienced while base level relinquishes control of the machine to pop its function call entries from the stack and is then re-entered at its fault entry before taking the reference return.

5.2.2.4 Sanity. The integrity of the system is preserved through a sanity detection mechanism implemented as a hierarchy of sanity timers, integrity checks implemented in native and emulated software, and a recovery mechanism implemented as a hierarchy of initializations.

Each application process has a built-in set of checks designed to detect any abnormal behavior that may lead to insanity. In particular, the TSPS process has preserved, through emulation, the hierarchical structure of sanity checks between base-level, low-priority J-level, and high-priority J-level. Also retained are the critical data structure checks and a monitor of the frequency of maintenance interrupts. The only sanity check that could not be emulated was the hardware long timer, previously implemented in the SPC 1A to check high-priority J-level sanity. This has been replaced by a software sanity timer implemented by AIM.

At the next higher level, the AIM process is responsible for monitoring the sanity of the TSPS process. It does so through the software sanity timer implemented as a counter shared between the two processes. This counter is incremented by high-priority J-level and decremented and then checked by AIM every 500 ms. The expected value of the counter is 0. Repeated non-zero values indicate an insane condition.

The sanity of AIM is in turn monitored by the DMERT System Integrity Monitor (SIM), a process responsible for the sanity of the entire software system. SIM monitors AIM, and implicitly the entire application, by an application sanity timer. AIM activates the application sanity timer after successful process creation in a system bootstrap. AIM must continue to indicate normal operation by sending a periodic sanity event to SIM to reset the timer.

The highest sanity check in the hierarchy is the 3B20D hardware

sanity timer. SIM periodically resets the sanity timer. Hence, the sanity of SIM is implied by the failure of the timer to fire. Should the timer fire, a stop-and-switch operation will be performed to force the off-line processor into the active role.

5.2.2.5 Overload control. The main objectives of the overload control strategy in TSPS No. 1B are to:

(*i*) Preserve system sanity

(*ii*) Maintain a high level of call completions regardless of the load applied to the system.

There are two types of resources that may be exhausted and lead to overload: TSPS call-processing resources, used exclusively by the TSPS process (e.g., TSPS peripherals, software resources used for call processing within the TSPS process, TSPS real time, etc.); and DMERT resources, either used exclusively by DMERT or shared with TSPS (e.g., kernel message buffers, nonswappable main memory, disk swap space, etc).

The TSPS No. 1 overload control strategy for TSPS resources has been preserved in TSPS No. 1B, with the software implementing it being emulated within the TSPS process. In this strategy, the E-E cycle time represents the detection parameter, while the number of calls being admitted and the number of active trunks represent the load control parameters. As the load increases, the E-E cycle time also increases in value. There are three thresholds for the E-E cycle time, determining the transitions from normal state, to phase 1 (minor) overload, to phase 11 (major) overload, and system initialization. During overload, the number of calls admitted and the number of active trunks are gradually reduced. As the load decreases, the E-E cycle time also decreases in value and the system is returned to its normal state.

The overload control strategy for DMERT resources shared by DMERT and TSPS is based on overload detection by the process that administers the particular resource, overload monitoring by the SIM process, and overload control by the SIM and AIM processes. Upon determining that an overload condition exists, SIM notifies the craft, attempts to free some resources to alleviate the condition, and then notifies AIM. For some DMERT-detected overloads (e.g., kernel or supervisor and user-level lockout) an initialization is executed to restore resources. Once notified of these overload conditions, AIM requests a DMERT Level 1 and TSPS minor audit initialization (MNA) initialization. Subsequent overload indications result in escalation of initializations.

5.2.2.6 Processor switch. An example of the communication between DMERT and the TSPS application required for effective control of the system is illustrated by the processor switch. The duplex SPC 1B

normally operates with one processor on-line actively processing calls and the other off-line with its memory continuously updated on every write operation. The off-line unit is thus ready to be switched on-line should the need arise. A processor-switch request can be one of three types: recovery, routine, and manual. A recovery processor switch is requested when a fault has been detected in the on-line processor and processing cannot continue on it. The routine processor switches are scheduled periodically, while the manual requests are made by the craft when needed.

Certain time-sensitive activities within the TSPS application, e.g., writing on the AMA tape, benefit from an advanced notification that a processor switch is about to take place. Such a notification is used to complete time-sensitive operations and get ready for the switch. However, only routine and manually requested switches can be postponed until the application has been notified and given its approval. Therefore, the routine and manual processor switch requests are routed through the AIM process. In turn, AIM notifies the TSPS application processes of an imminent processor switch and, after their completion of time-sensitive operations, returns a switch go-ahead to SIM. As a defense mechanism, if the application does not approve the switch within 10 seconds, DMERT will proceed with the switch.

5.3 Other TSPS application processes

In addition to the TSPS process numerous other native-mode processes were developed to support the emulation. These processes provide needed functions for a complete system under DMERT. The major processes are discussed below. Figure 13 illustrates the interaction between these major processes. The AIM process has been described previously, and therefore will not be included here. The PSI Diagnostic Driver and Control processes are detailed in Ref. 6.

5.3.1 Kernel processes

5.3.1.1 PSI diagnostic driver. The PSI Diagnostic Driver process is a killable kernel process that performs diagnostic tests of the on-line PSI. To perform the on-line tests the driver must synchronize its operation with TSPS J-level operation in order not to interfere with ongoing peripheral I/O. It will execute the tests at execution level 12 when a non-interfering 3-ms window between J-levels is found. Its other processing is performed at lower execution levels. The driver shares memory with AIM and the TSPS process for synchronization of activities and in order to monitor TSPS peripheral equipment equippage and status.

5.3.2 Supervisor processes

5.3.2.1 File system interface. The TSPS No. 1B application has only

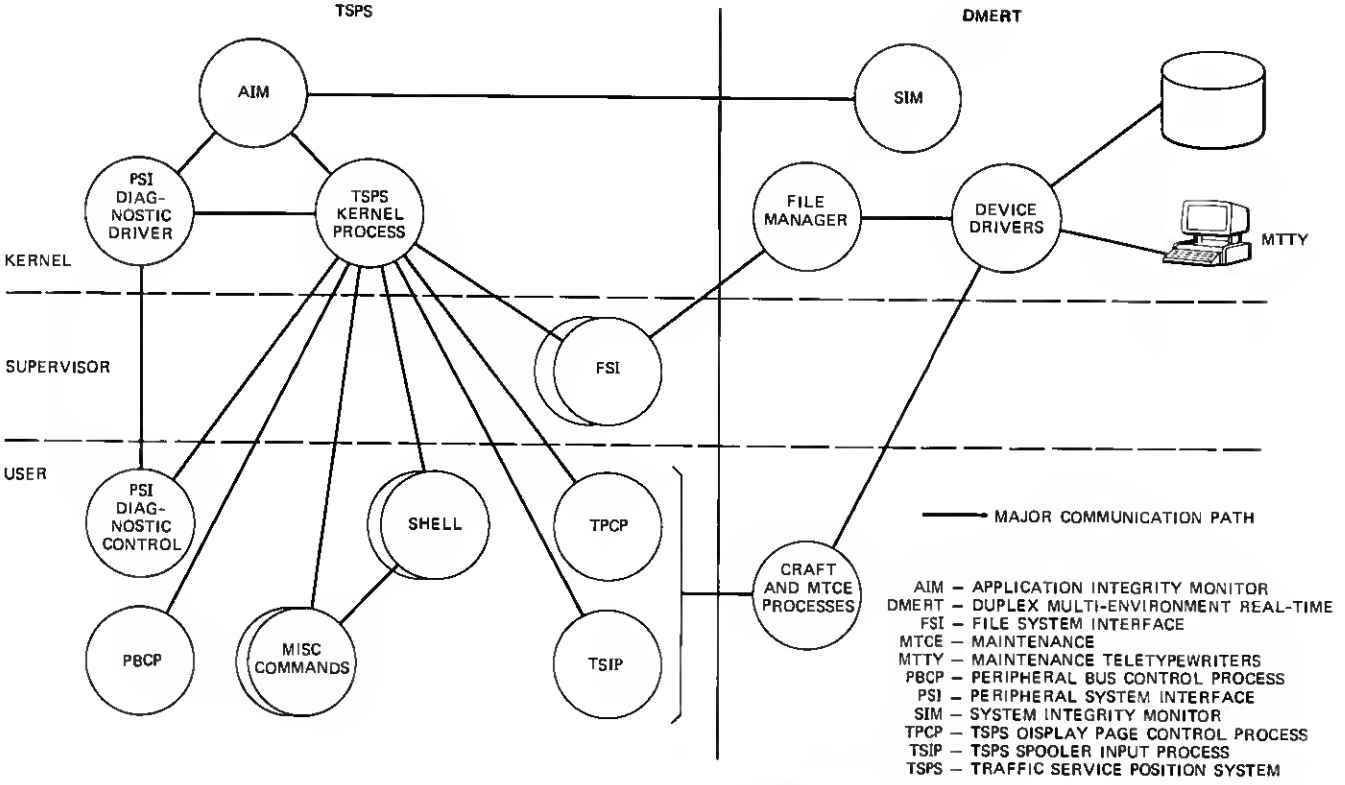

Fig. 13—Interaction between TSPS and DMERT processes in TSPS No. 1B process architecture.

one supervisor process, the File System Interface (FSI). The SPC 1A's program tape unit (PTU) was not retained in TSPS No. 1B. All required operations of the PTU were functionally reproduced with the FSI. Examples of such functions include dumping emulated office data, protected memory, or program to tape. The FSI shares the emulated-code address space and interfaces with the DMERT file manager to transfer data between the emulated address space and the DMERT disk file system. The 3B20D's nine-track tape unit and DMERT I/O facilities are used to move the data between tapes and the disk file system. None of the FSI functions are time-critical. Hence, the FSI is designed as a killable supervisor process that is created upon demand and dies when it has completed its function. Multiple instances of the FSI can simultaneously coexist as long as their respective functions do not interfere with each other.

5.3.3 User processes

The TSPS application has numerous user-level processes. Most of these perform demand tasks as the result of craft input. The process is created to perform the task and then dies when it is completed. The most significant user-level processes are described below.

5.3.3.1 Craft interface processes. The maintenance center craft interface for the TSPS No. 1B consists of a video display terminal for input, output, and status displays, along with an adjacent printer for a hard copy of all output messages. The terminal's screen is split into four areas, as shown in Fig. 14. The top is for system status information and is always displayed. The variable-sized middle section is for displays which, for example, may show the status of a particular hardware subsystem. The remainder of the screen is for scrolling system output messages, and there is a single, dedicated line for input. This interface is also remoted to the Switching Control Center System (SCCS), described in Ref. 9, for remote maintenance.

Although the actual I/O to the devices is done by the DMERT I/O driver, the data read and written to the devices are formatted and interpreted by a collection of user-level processes. The major DMERT processes include the input Shell, the Controller of Output Spooler Process (CSOP), and the Display Administration Process (DAP).

The application interface to CSOP and DAP is designed to be from other user-level processes. Hence, TSPS has developed two user processes that serve as intermediaries between the TSPS kernel process and CSOP and DAP. The TSPS Spooler Input Process (TSIP) shares a memory buffer with the TSPS process. TSPS queues messages by priority in this buffer. TSIP dequeues these messages and passes them to CSOP via interprocess messages. CSOP then merges the TSPS-generated messages into a single, systemwide, prioritized queue for

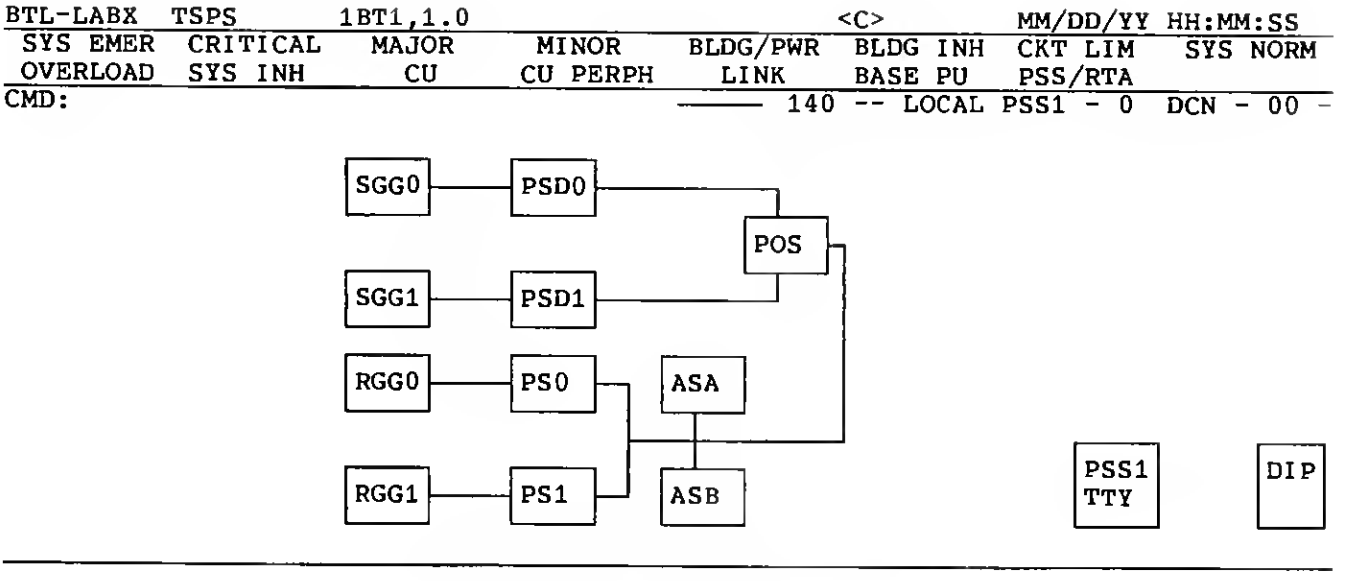

Fig. 14—Maintenance center craft interface for TSPS No. 1B.

printing to the maintenance terminals. The TSPS Display Page Control Process (TPCP) controls the TSPS application displays. TPCP receives change of state information from the TSPS process via interprocess messages. TPCP then translates this into the appropriate display page control information and directs DAP via interprocess messages to change the state of the various display page indicators.

The above-mentioned processes are potentially subject to being killed as the result of a system error or manual action. However, they provide a critical service and must have continuous operation. DMERT provides a Craft Monitor (CMON) process that monitors the operation of craft processes. CMON will immediately recreate any craft process that dies, along with any other process that must be reinitialized. CMON itself is monitored by the User-Level Automatic Restart Process (ULARP). ULARP will automatically restart any user-level process under its surveillance should it die. ULARP execution is monitored by the DMERT System Integrity Monitor process (SIM).

TSPS input message syntax is quite different from that used by DMERT. The processing of these input messages is also performed quite differently. As such, it was necessary to modify the DMERT shell to handle TSPS input as well. When an input message is entered by the craft and read by the combined shell, it first checks for a TSPS syntax message (the most likely to be input). If the input is a TSPS message, it is sent to the TSPS process in an interprocess message. The TSPS process will look up the message in its memory-resident catalog and act on it accordingly. If the input was not a TSPS message, the combined shell handles it in the standard DMERT fashion. That is, a disk directory search is performed to find the appropriate user-level process to execute in response to the command. This process is then created and executed to handle the input accordingly.

5.3.3.2 PSI diagnostic control process. The PSI diagnostic control process is a user-level process that controls the execution of diagnostic tests on the off-line PSI. The PSI diagnostic was designed similar to DMERT common-system diagnostic processes and is woven into the DMERT diagnostic control structure. It is reacted and executed on demand as a result of a manual request, routine exercise or automatic diagnostic request. The PSI diagnostic creates the PSI diagnostic driver and communicates with it through interprocess messages to coordinate on-line and off-line PSI tests and TSPS process F-level interrupt recovery actions.

5.3.3.3 Peripheral bus control process. Power to the PSI peripheral buses is controlled by the 3B20D Processor's scanner/signal distributor (SC/SD). The PSI bus-power control points are duplicated with a set on a SC/SD controller in each I/O processor (IOP). The Peripheral

Bus Control Process (PBCP) monitors and controls the status of the PSI peripheral buses. PBCP interfaces with the DMERT SC/SD administrator and the TSPS process via interprocess messages to coordinate actions. PBCP is a critical process that must always be active. Hence, it runs under the surveillance of ULARP.

VI. SUMMARY

This article has described how the characteristics of the TSPS No. 1 and the 3B/DMERT systems have been combined in the TSPS No. 1B architecture. Successful emulation of the TSPS No. 1 software has been accomplished by hardware, firmware, and software in the TSPS No. 1B. Emulated code, along with associated native code, has been structured as a single kernel process running under DMERT. This kernel process cooperates with other application processes and DMERT to form integrated maintenance and craft interface packages. The TSPS No. 1B provides a modern, flexible vehicle for the future expansion of TSPS services.

VII. ACKNOWLEDGMENTS

The TSPS No. 1B software described in this article was the result of contributions by many people. The authors wish to acknowledge their help in the preparation of this article. In particular, the authors wish to thank J. M. Aiken, P. S. Bogusz, D. L. Brown, D. L. Hofmockel, M. H. Richardson, E. S. Sachs, and M. D. Soneru for their contributions.

REFERENCES

1. R. E. Staehler and J. I. Cochrane, "Traffic Service Position System No. 1B: Overview and Objectives," B.S.T.J., this issue.
2. B.S.T.J., *49*, No. 10 (December 1970).
3. B.S.T.J., *58*, No. 6, Part 1 (July-August 1979).
4. "3B20D Processor & DMERT Operating System", B.S.T.J., *62*, No. 1, Part 2 (January 1982).
5. D. L. Bayer and H. Lycklama, "*UNIX* Time-Sharing System: The MERT Operating System," B.S.T.J., *57*, No. 6, Part 2 (July-August 1978), pp. 2049–86.
6. H. A. Hilsinger, J. H. Tendick, R. A. Weber, and G. T. Clark, "Traffic Service Position System No. 1B: Hardware Configuration," B.S.T.J., this issue.
7. T. Hack, T. Huang, and L. C. Stecher, "Traffic Service Position System No. 1B: Software Development System," B.S.T.J., this issue.
8. B. A. Crane and D. S. Suk, "Traffic Service Position System No. 1B: Capacity and Reliability Evaluation," B.S.T.J., this issue.
9. J. J. Bodnar, J. R. Daino, and K. A. VanderMeulen, "Traffic Service Position System No. 1B: Switching Control Center System Interface," B.S.T.J., this issue.